U S L / D B M S   N A S A / R E C O N

W O R K I N G   P A P E R   S E R I E S

Report Number

DBMS.NASA/RECON-21

(NASA-CR-184529) KARL: A KNOWLEDGE-ASSISTED RETRIEVAL LANGUAGE M.S. Thesis Final Report, 1 Jul. 1985 - 31 Dec. 1987 (University of Southwestern Louisiana. Lafayette. Center for Advanced Computer   N89-14569

Unclas
G3/82 0183567

For more information, contact:

Wayne D. Dominick

Editor
USL/DBMS NASA/RECON Working Paper Series
Center for Advanced Computer Studies
University of Southwestern Louisiana
P. O. Box 44330
Lafayette, Louisiana 70504

KARL: A KNOWLEDGE-ASSISTED RETRIEVAL LANGUAGE

A Thesis

Presented to

The Graduate Faculty of

The University of Southwestern Louisiana

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Spiros Triantafyllopoulos

Fall 1985

KARL: A KNOWLEDGE ASSISTED RETRIEVAL LANGUAGE

by

Spiros Triantafyllopoulos

APPROVED:

Wayne D Dominick

Dr. Wayne D. Dominick
Associate Professor
Center for Advanced
Computer Studies

[illegible]

Dr. Lois M. L. Delcambre
Assistant Professor
Center for Advanced
Computer Studies

[illegible]

Dr. William R. Edwards
Associate Professor
Center for Advanced
Computer Studies

Dr. Joan T. Cain
Dean, Graduate School

Date: November 18, 1985

ACKNOWLEDGMENTS

I wish to express my appreciation to my committee chairman and advisor, Dr. Wayne D. Dominick, for his time and effort during the development of this thesis. His support throughout my graduate studies has been invaluable. I would also like to thank my committee members, Dr. Lois M. Delcambre and Dr. William R. Edwards, for helpful suggestions during the preparation of the thesis.

The National Aeronautics and Space Administration (NASA) deserves my appreciation for the funding I received as a Research Assistant from January to December 1984, under NASA Contract Number NASW-3846, "NASA/RECON: Course Development, Evaluation and Administration".

Mr. Patrick Landry and the staff of the Computer Research Laboratory deserve my appreciation for technical advice, suggestions and valuable discussions. My fellow graduate students I-Hsiung Liu, Carlos Pena, Suzy Gallagher and Frank Chum made useful comments on this work. Finally, special thanks go to my fiancée, Jinous Bassari, for her support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

KARL: A KNOWLEDGE-ASSISTED RETRIEVAL LANGUAGE

CHAPTER 1

INTRODUCTION

Availability of computer resources, reduced initial and operating costs, and simpler operating procedures have all contributed to the introduction of computers into a wide variety of applications. As the user community expands, the number of non-computer literate users also increases. While, in the installations of the early years, computers were "viewed only through glass doors and had their own white-robed therapists" [Kidder 82], many of today's users neither possess the knowledge necessary to use the computer efficiently nor are they willing to obtain a working knowledge of applications software, due to time and other constraints.

Currently, computers are increasingly used in more aspects of human life than ever before. As a consequence, more and more humans come in contact with the computer, sometimes viewing the machine as the panacea that will solve all their needs. More realistic users, however, while aware of the computer's capabilities, sometimes do not possess skills for effective communication with the computer, or are not able or willing to acquire these skills. The so-called "casual users", that constitute a significant number of total users, use the computer in a wide variety of applications, and sometimes suffer the consequences of training and retraining for each application. Such applications range from financial modelling to data communications and from statistical analysis to word processing.

The application that this thesis will concentrate upon is data retrieval from a database. Data retrieval represents a major activity of computers; still, few systems offer efficient, user-friendly interfaces. Various command languages, most frequently known as query languages [Ullman 82] and other systems have been developed [Date 83], but still, for the average user, such systems require a major commitment if they are to be utilized properly.

The main problem in the communication process between the user and the system is the "Knowledge Gap", that is, the contrast between the knowledge that the user has to gather in order to use the system and the system's inability to obtain and use knowledge possessed by the user regarding the particular application. The "knowledge gap" is present in cases when the user has to learn about the system while the system is unable to either obtain or use knowledge about the user.

A recent survey of people's attitudes towards computers indicates that many candidate users believe that computers can solve any problem with very limited human interaction [Morrison 84]. Such ideas were introduced by early scientific predictions and even science fiction [Clarke 71]. According to these forecasts, the "almost human" computer communicates with humans in English. Having been exposed to such ideas, future users find it difficult to adjust to the existing technology that usually does not adapt to them and, as a result, do not use the computer to its full potential.

Such predictions, however, indicate that English is the most efficient way of communicating between a non-computer expert and the computer. In the field of data retrieval, there have been several programs that allow user-machine interaction in restricted English, with excellent results [Mylopoulos 76; Eisenberg 84]. The application of computer science areas such as Artificial Intelligence, as well as of interdisciplinary sciences such as Cognitive Psychology and Computational Linguistics has produced new methods that improve the data retrieval process to a large extent. This thesis will present such a system, the Knowledge Assisted Retrieval Language (KARL), that attempts to provide a solution to the problem of man-machine interaction during the process of retrieving data from a database.

CHAPTER 2

BACKGROUND

2.1 THE COMMUNICATIONS HIERARCHY

Communication between a user and the computer can be performed on several different "levels", depending on the user skills, the computer system available, and the task to be performed. Traditionally, these levels are defined as increments, from machine to user convenience, as in Figure 1.

```
      Restricted Natural Languages (PLANES, INTELLECT)
                              : : :
                              V V V
   Special Problem Oriented Languages (Minitab, Simscript)
                              : : :
                              V V V
    Problem-Oriented Programming Languages (Prolog, LISP)
                              : : :
                              V V V
 Procedure-Oriented Programming Languages (PL/1, Ada, BASIC)
                              : : :
                              V V V
          Machine-Oriented Languages (Assemblers)
                              : : :
                              V V V
                      Machine Languages
```

Figure 1 Computer Language Hierarchy

The programming languages mentioned in Figure 1 represent a variety of uses; some are for general-purpose programming, while others are for a variety of problem-oriented applications or even very specific applications such as data retrieval or simulation. Some overlapping of tasks has been taking place, i.e., the case where a language is used for applications other than the ones for which it was designed, but typically, task separation according to functionality is well defined in the hierarchy [Pratt 83].

2.2 EARLY DATA MANIPULATION TECHNIQUES

In the early data management systems, retrieval was performed by programs written in general-purpose languages of the time, the most popular being COBOL and FORTRAN IV [Date 81]. Such systems were rather crude for today's standards, since they provided none of the characteristics of "modern" software, i.e., reliability, ease of maintenance, portability, modifiability, etc., as defined in software engineering texts [Turner 84; Sommerville 82].

As demands for flexibility and performance increased, more sophisticated file management systems were developed that allowed subroutine libraries for common code segments to be maintained and offered some type of protection and sharing [Ullman 82]. Such systems provided some facilities for automated organization of data in tabular forms, typically through a flat-file model. Still, reorganization of the information present, full protection, and data independence was not provided. Such systems, known typically as File Management Systems (FMS), provided facilities for definition, access and update of indexed files through hashing tables, B-trees, sequentially or via indices [Wiederhold 77; Theorey 83].

2.3 DATABASE MANAGEMENT SYSTEMS

Since user demands for improved computer-based information systems were continually increasing, Data Base Management Systems (DBMS) were introduced, initially for mainframes and later for smaller systems [Stonebroker 76; Date 81; Date 83]. DBMS's offer considerable advantages compared to file management systems, primarily in the areas of integrity, flexibility, and security. Even in their early forms, they provided sophisticated facilities for definition of data through data models, access control through locking at the database, file, record, or even field levels, data integrity through language constructs, and reorganization (restructuring) of data already in the database. Furthermore, many DBMS's developed were portable in the sense that they were designed for a variety of environments, and not for a specific environment.

In the area of data definition, virtually all DBMS's provide a_facility for defining structured data that reflect the representation of user information in the database. Such representations were made based on the database type that specified the major organization of information in trees, tables or networks. The representation of data, as well as the allowed manipulation of data contained in the database, depends on the way the information is organized. This is especially true in the case of interaction between the user and the machine: the type of database that is used specifies the allowable operations, which in turn specify the form of user requests that have to be issued in order for a particular task to be accomplished. There are three major design philosophies for DBMS, that resulted in three distinct DBMS models, as defined in [Ullman 82]:

(1) Relational Model: Based on mathematical set-theory and domains/ranges, the relational model uses tables to store data. It uses a collection of constructs known as the schema to indicate the grouping and relationship of data. The relational model has operations defined that match the operations present in set manipulation, as well as operations that are used in traditional storage and retrieval of data.

(2) Network Model: Using binary, many-to-one relationships, this model represents data in simple directed graph forms. The network model also groups data entities into

sets, but the set operations are more explicit than the corresponding ones on the relational model. The network model, as defined by the ANSI/SPARC DBTG standard [Date 81], provides set concepts such as "owner", "member", "set type", and others, for expressing the relationship between data objects. Initially, one-to-many type relationships were supported, but with the creation of intersection records and other techniques, many-to-many type relationships can be represented.

(3) Hierarchical Model: This model represents a "forest" network, where only parent-child relationships are allowed. The hierarchical model is the oldest of the three models of database systems. In the hierarchical model, a tree-like structure is defined, with a one-to-many (but not many-to-many or many-to-one capabilities that are found in the network model) organization of data into database records.

To create a database, a set of descriptors has to be created. These descriptors indicate the names, types, and other attributes of data objects and the relationships between the data objects created. These descriptors for each record constitute the database schema. Subsets of the database can also be considered, yielding sub-schemas. The terminology depends on the type of database management system that is being used to process the schema, but the concept of database creation is similar in all models.

After a database is created, it is typically initialized from data that has already been collected, or transformed from previous files. Then, the entire system is turned over to the users that are going to be using the information stored. While such a method of operation is not always followed, it represents fairly closely the "modus operandi" of the database environment.

2.4 ACCESSING THE DATABASE

To access stored data, typically a request has to be issued to the database. This system-dependent request, called a query, has numerous forms, the most common ones resembling programming languages [Epstein 79]. The user has to either learn the query language or use the database through a simplified, application specific interface through a program or command file. There is also the alternative of using application programs that invoke the database facilities for creation and manipulation of data via existing programming languages, through embedded code, like EQUEL and Ingres, or through subroutine calls, like the MRDS system and its associated "dsl_" calls [Honeywell 76]. Finally, non-language alternatives exist in the form of questionnaire-style forms, completed interactively by the user, like the IBM

Query-By-Example system, menu-driven systems such as DBaseII, and others [Zloof 76; Date 83].

2.5 NATURAL LANGUAGE DATABASE QUERYING

The hierarchy that was mentioned earlier can be abstracted to two main categories, depending on the orientation of the languages. Figure 2 shows the generic classification.

NON-PROCEDURAL, PROBLEM ORIENTED LANGUAGES
NL's, Object-oriented, problem-oriented.
Little requirement of computer science knowledge; knowledge of application required

PROCEDURAL, GENERAL-TYPE LANGUAGES
Most programming languages. Require knowledge of basic computer science skills, but no application knowledge.

Figure 2 Language Orientation

Given a certain task, there is a collection of metrics, both software and user-related, that can be applied to the language that is used to determine its relative efficiency (in terms of both human and machine effort). For example, assume a collection of data exists about airline flights, with the schema as illustrated in Figure 3. The database model shown is relational, but the terminology and metrics can be used with other database models.

Flight Number	From City	To City	Depart Time	Arrive Time	Flight Cost
114	LFT	NOR	1100am	1145am	45.00
023	LFT	DFW	1040am	1240pm	78.40
112	DFW	DET	1245pm	0220pm	120.00
122	NOR	MEM	0320pm	0410pm	94.00

City Code	City Name	Airport Phone
LFT	Lafayette	2341344
NOR	New Orleans	5789894
DFW	Dallas-Ft.W	5872565
DET	Detroit	7642334
MEM	Memphis	2223443

Figure 3 Sample DBMS Schema

The above example is adopted from [Date 83]. In order to perform data retrieval using a general-purpose programming language, as defined by the hierarchy presented earlier, a full program would have to be written, compiled, tested and run before the application can be created. Then, the program created will be only for a very specific application, and new applications will result in more coding. Although the size of required code would vary, depending on the language, a natural language (NL) query would typically be much smaller than a program writen in a programming language. The natural language query shown in Figure 4 is even shorter than its corresponding formal query.

NATURAL LANGUAGE QUERY:

```
  show the flights that leave from Lafayette to Dallas
   before 1100pm.
```

FORMAL LANGUAGE QUERY:

```
  retrieve flight where
   (flt.from = retrieve city.code where city.name="Lafayette")
     and
   (flt.to   = retrieve city.code where city.name="Dallas")
     and
   (flt.depart lt "1100pm")
  print flight
```

Figure 4 Natural and Formal Language Queries

Modern software methodologies [Freeman 81; Brown 76] suggest that any application can be implemented using any computer language as the implementation vehicle; a request for the retrieval of data from a collection of data can be implemented in almost any conceivable computer language. Human efficiency in the retrieval process, especially as human time becomes more critical, should be an important future research metric.

2.6 MAN-MACHINE INTERACTION CONSIDERATIONS

In Chapter 1, the case of the "casual" user was mentioned as the important issue to consider, since the casual user represents a large percentage of total information system usage. As [Dillon 83] reports, computers have been introduced to a large number of non-computer skilled users who were eventually expected to obtain computer literacy in order to fully use the machine's capabilities. However, evidence of improvement of computer literacy has not become apparent, in spite of the wide introduction of computers in many household- and school-level activities such as Computer-Aided Learning and games.

The focus of this research is the retrieval of information from on-line databases. Casual users are the most frequent users of such systems. Typically, casual users perform relatively simple and/or routine retrieval tasks. Such tasks, however, still require system-specific knowledge to be acquired in the form of invocation procedures, command languages, query formatting and execution, etc.

Since most casual users are neither able or willing to obtain the necessary knowledge in order to use a database system effectively, the opposite direction, i.e., enabling the system to obtain detailed knowledge about the application area(s), appears to be a way to bridge the "knowledge gap" between the user and the machine. Efforts in the area of user/machine interfaces have produced remarkable results. For example, the User-Derived-Interface (UDI) reported in [Good 84] has capabilities of obtaining knowledge from specific users and applications, and incorporating the knowledge obtained through interactive use for later utilization.

It is apparent that the knowledge gap can be bridged only by supplying the software system with user-derived knowledge, that is, knowledge from the user's point of view regarding the particular application area(s) that the database contains. Several successful natural language front-ends exist for commercially available DBMS's, and more are developed as prototypes of front-ends to other applications, using similar techniques for translating user NL requests into system-specific requests. However, the main problem of such configurations, and in particular in the field of database systems due to the variety of applications, is the inability of the system to retarget to different applications.

The result of such characteristics is a system that performs acceptably on a particular domain, but requires considerable "surgery" in order to adapt to a new environment. The PLANES system is one such example of a system [Wassermann 85]. Desiged originally to maintain a database of airplanes and their associated service and flight records, PLANES performed well in a near-production level. Its "application knowledge", however, was encoded in the source code, thus necessitating recoding for use in other application areas.

2.7 NATURAL LANGUAGE DATABASE FRONT ENDS

The rationale for a natural language interface for a DBMS is simple: increased efficiency of the man-machine interface through improved communication capabilities [Mylopoulos 76; Good 84; Salton 83]. Another example will demonstrate the simplicity of natural language queries, from the casual user's point of view.

The query of Figure 5 retrieves salaries and names of all male employees with salaries more than 18,000 dollars. The first query is for the INGRES relational database system [Stonebroker 76; Epstein 79]. An equivalent natural language query is also presented in Figure 5.

```
FORMAL QUERY

     RANGE OF E IS EMPLOYEE
     SELECT (SALARY, NAME)
     WHERE (SALARY > 18000 &
            SEX = "MALE")
     PRINT E

NATURAL LANGUAGE QUERY

     PLEASE PRINT THE NAMES AND SALARIES OF ALL MEN
     THAT EARN MORE THAN $18,000 A YEAR
```

Figure 5 Formal vs. Natural Language Query

Simplification of the user interface with the database results in several improvements to the process of communication between users and computers. The results that are expected typically include:

(1) Increased productivity, since the users of the database(s) will no longer be required to formulate queries in a non-native (i.e., formal query) language.

(2) Better system utilization, since users of a NLQS are expected to be less prone to make syntactic or semantic errors (using their own natural language, rather than an unfamiliar formal query language). Such errors can result in having users re-issue queries without being sure that they are correct (trial-and-error approach). System utilization in such cases is thus reduced due to having users "experiment" in order to perform their retrieval tasks. Also, a reduction in the amount of training time required can make more user time available for productive vs. non-productive work [Logsdon 76].

(3) Reduced user frustration, since the communication process is performed in the user's terms rather than in the system's.

(4) Virtual elimination of a training period. However, a brief introduction to the system's capabilities and associated features (i.e., how to "teach" the system new words, use customized output formatting, etc., if such are implemented) would be required.

(5) Simpler query structures by allowing the storing and retrieving, as needed, of expressions or characteristics in the appropriate grammatical forms.

(6) Improved handling of concepts that appear "natural" to a human user, such as thesaurus and dictionary support, or even cross-referencing between records [Salton 83].

(7) No need for retraining for new applications or updates of current applications will be needed. Also, a NL DBMS interface can be a part of an integrated NL-based front end for a variety of applications which all have NL front-ends, thus elimininating the need to learn several different command procedures [Green 76; Coombs 76].

2.8 EXISTING NL SYSTEMS AND PROBLEMS

There are numerous systems that provide a natural language database front end [Grishman 84]. However, detailed case studies of such systems have indicated a series of characteristics that are not desirable. Development time can be long, even spanning a period of years. Complexity appears to be a main factor. The task of understanding natural language (even in restricted forms) is non-trivial; development of NL systems has traditionally been extremely consuming in man-power and resources.

Other systems suffer from being tailored to a restricted application domain, and are thus not adaptable to new applications. For example, the BASEBALL or the LUNAR systems, mentioned in [Wasserman 85], while capable of handling relatively complex queries, could not be ported to other application domains without major revisions. Knowledge was essentially "hard-wired" to the effect that modifications to the source code needed to parse and/or verify the NL query would be needed, if the application domain were to change.

Another problem is portability between different computer systems and/or supporting software tools (such as DBMS hosts, languages, operating systems, etc.). This is also often true in the case where a NL system is developed in a research environment, with a "toy" database being supported. The NL system, if it is to be useful, must be able to interface successfully with existing prototype DBMS's or commercially available DBMS's.

A number of NL systems have been successful in their respective domains; some will be briefly presented below in order to demonstrate the current state-of-the-art in NL systems, as well as the general techniques that have been followed in their implementation. The majority of the systems that will be discussed are experimental systems while one has been in production use for several years.

One of the early NL front ends for database systems is the PLANES system. PLANES was developed as a front end to a large database containing maintainance and flight records for Navy planes. Its main structure was an ATN-based parser that constructed the network after analyzing sub-patterns (subsets of the entire sentence, known as semantic constituents). As the application domain as well as the underlying database structure were fixed and not subject to changes, the ATN parser was "hardwired" with application-specific knowledge. Such knowledge enabled the parser to determine the specific semantic constituents and, following a continuous left to right scan, determine the entire NL query structure and create the ATN. The semantic constituents understood by the parser were fixed and related to the application only; they included phrase terms related to time periods, aircraft types, flight and malfunction codes and identifiers, maintainance actions, and other application-specific phrase segments.

PLANES was able to parse and process a number of English language constructs. It was also capable of identifying and processing NL queries expressed in statements which did not follow exact syntactic rules. Being more semantic than syntax oriented, PLANES would ignore much of the underlying grammar of a sentence and use only semantic information present in the form of the semantic constituents. Implementation-wise, PLANES followed a rather inefficient push-down automaton mechanism that would often push unacceptable parts of a sentence for matching without being able to efficiently recognize return conditions [Tennant 81].

One capability that PLANES helped demonstrate as feasible was its handling of pronoun references and several types of elliptic queries. If a semantic constituent required for a query was missing (such as a time period or an aircraft type) a look-up in a previous query list could retrieve the missing part(s) of the query and process it properly. Its elliptic and pronoun handling capability can mainly be attributed to its restricted application domain and "hardwired" knowledge and database structure. PLANES proved a rather usable system within its application domain.

RENDEZVOUS is another NL front end designed specifically for relational database systems, taking into consideration the database schema and the processing of sub-queries in order to produce the formal query from the user's input. In addition, RENDEZVOUS did not follow other systems in performing a single transformation of the input NL query into an intermediate representation and then into the formal query, but rather followed a number of production rules at different stages of the query processing.

RENDEZVOUS is implemented as a semantic grammar system implemented thru production and transformation rules [Tennant, 81; Wasserman, 85]. Repeated application of the rules would then transform the input NL query into the formal query. The rules themselves would be implemented as pairs of constructs; each rule would have a left side (LHS) and a right side (RHS). Pattern matching would match a subset of the input query into an LHS, and then the LHS would be replaced by its corresponding RHS. If no match was obtained, the next LHS would be tested. In addition, boolean expressions could be included as LHS components, thus allowing conditional replacement. Finally, the RHS could contain a function call to be executed and the result placed as the LHS replacement (such as date, location, etc.).

RENDEZVOUS was able to initiate clarification dialogues with the user if additional information to process the input queries was required. In most cases there was no real conversation between the NLQS and the user, but rather a multiple-choice type interaction where the program would display the possible interpretations and the user would be prompted to select one. In other cases (such as misspelled words) the user would be prompted to key in the word again. Finally, if the conceptual information presented in the query was incomplete, the program would prompt for additional statements instead of processing the query immediately. Then, whatever the user typed as part of a continuation dialog either was added to the current query or replaced statements that already existed.

Several rule classes were provided for the processing of input queries; however, little flexibility was provided for adding rules to the rule base or modifying existing rules. Rule classes were applied one-at-a-time, with no heuristics being used for efficiency. When a rule matching a LHS was found, its production was applied and the RHS replaced the sub-construct. The procedure would then be repeated as necessary. If sub-queries generated as parts of LHS expressions failed, the system would indicate the sub-query that failed and prompt for further action.

Although not as sophisticated as other NL systems, RENDEZVOUS introduced several new concepts in the field of NL processing. Concepts such as continuous dialog between the user and the system, transformations using rules and query failure analysis have then been used by other systems. Also, interface of the NL query processor with an existing relational database system (as opposed to interfacing with "toy" databases for other NLQS's) is important for production-level systems.

The third system to be discussed, INTELLECT, is a commercial product marketed for interface with existing database systems in a variety of applications. It is capable of interfacing with different database systems that exist already, under a variety of underlying data models.

INTELLECT is a commercial product and as such, information regarding its internals and implementation has not been readily available. It is a very sophisticated production-level system capable of supporting different applications, capable of user-defined term processing, user-defined or application-based query output formatting, and others [INTELLECT, 85]. The main concept that is present in INTELLECT is the system dictionary, or "lexicon". Different applications can be included in the system's capabilities by creating new application lexicons, populating the lexicons with the initial knowledge required to process typical user queries, and then releasing the system for production use. As mentioned earlier, the system has a learning capability that allows users to include their own terms and idioms. Also, custom formats can be provided for output formatting where applicable.

Development of INTELLECT required several years. Also, the requirements for using it are rather demanding, restricting its use only to mainframe-based systems. INTELLECT's run time requirements include the PL/1 resident and transient libraries, and a number of resident utilities for creation and maintainance of lexicons. Finally, the cost of acquiring INTELLECT is large, when compared with the cost of other software systems for similar environments. A binary license for INTELLECT can cost as much as $67,000. Although the retrieval of data is improved by using such a system, creation and maintainance of the application-specific lexicons requires the use of special analysts (typically knowledge engineers), thus increasing the operational costs even more.

INTELLECT is oriented more towards MIS applications. Additional software available for use with it provides capabilities for NL based graphics, NL lexicon construction, and others. INTELLECT's capabilities for processing natural language queries include extensive pronoun reference capabilities, ambiguity and ellipsis handling, interactive dialogues with the user for clarification or requests for additional information, and others. INTELLECT's main advantage is its use of the lexicon that allows different applications to be mapped on lexicons and then using the lexicons for retrieval.

INTELLECT is more word-driven than the previous two systems discussed that were more semantic and concept-driven. This results in reduced semantic verification capabilities. Also, database semantics information is not fully represented in the lexicon, thus reducing even more its semantic capabilities. Further, access for lexicon modifications is granted to all end users for lexicon updates, instead of providing personalized dictionaries. Neverthless, INTELLECT is a vast improvement in the area of user-oriented retrieval languages, in particular within an area where few of the recent advances in software design have been introduced (corporate MIS and data processing environments).

The last NLQS that will be discussed is one of the earliest approaches in NL front ends for databases [Tennant, 81]. The Airline Guide, developed in the late 1960's, had a number of interesting features that formed the basis for further research in the area of NL processing. Such concepts included improved semantic capabilities, and separate database and natural language systems (in contrast to other systems of the time that provided a common database/NLQS system, often with a "toy" database). Finally, the interface of the Airline Guide with the actual file management system that maintained the flight information was achieved at the formal query level (i.e., the Airline Guide would generate formal queries) so that portability to other applications could be facilitated.

As there were at that time no database systems in the form known today (i.e., relational, network, etc), the Airline Guide interfaced with a flat-file based system that maintained a machine-readable form of information about commercial flights. Only one record was provided for each flight. Most of the design efforts were concentrated on semantics, and although a syntactic analyzer was present, its functionality was reduced to the single task of providing the semantic analyzer with parsed sentence fragments.

Semantic analysis was the main focus of the Airline Guide. Its development timeframe (late 1960's) was immediately after the studies on semantics performed in the early 1960's with the task of selecting a semantically correct sentence out of a number of different syntactic representations. Semantic information for the Airline Guide was provided directly from the contents of the flight information file, without a separate dictionary or similar construct.

The Airline Guide used a traditional parse tree for the representation of the input sentence. The tree was constructed by the parser and was verified by comparing it to a collection of primitives that existed in the flight file. Such primitives were considered as functions relating user input words and terms used in the flight file. Predicates were also used to test conditions among functions. For example, a primitive function "CONNECT" would return the value of "true" or "false" if the parameters specified (that is, city names) were connected by a flight. Semantic information was collected for four different classes of words (nouns, noun modifiers, determiners and verbs) and the semantic information was used to build the actual formal queries. The Airline Guide allowed constructs such as quantifiers (explicit or implicit) to be used, thus expanding the vocabulary even more.

Although semantics-oriented, the Airline Guide did have a parser and a syntactic verification system. For each input query, its parse tree would be constructed and then the tree would be compared to templates provided. Once a template was provided that matched the supplied sentence, semantic analysis could proceed. Limited syntactic capabilities, however, result in loss of flexibility that is otherwise obtained by processing sentences which can be syntactically incorrect (for a given parser) but semantically correct, as is the case for "pidgin English" queries. Also, "hardwiring" the system vocabulary in the program reduces its flexibility and portability to new applications. Despite these problems, the Airline Guide was one of the first programs to demonstrate the feasiblity of NLQS's, and also to accept and implement query semantics as the main issue in NL query processing.

Concluding the overview of experimental and commercial natural language database front ends, some of the problems associated with NL query processing identified earlier in this sub-section can be visualized. Problems ranging from inefficient systems (PLANES had an average processing time per query on the order of 68 seconds) to inflexible systems that could not be ported to different applications, to systems that perform well on mainframe environments with support personnel but are unsuitable for mini- and micro-computer applications (INTELLECT) have long been known among researchers. However, most NL systems offer considerable improvements to the interface problem over the more traditional formal query systems, and their use has been demonstrated to be feasible [Blanning 84; Mylopoulos 76]. This is true, even considering the drawbacks mentioned that would be potential problems. Simply stated, the advantages outweigh the disadvantages [Tennant 81; Grishman 84].

2.9 GENERIC DESIGN OBJECTIVES

Based on the advantages and disadvantages of Natural Language Query Systems (NLQS), general objectives that can apply to any software system, and thus also be adaptable for a NLQS, are presented here. The generic objectives of the proposed design, the Knowledge Assisted Retrieval Language (KARL), include the following:

(1) Adaptability to new applications: The system should be able to adapt to new applications with modifications to the application-specific knowledge only, and no modifications to the system source code. The degree of adaptability (i.e., the spectrum of applications that

the system can handle without code modifications) would also depend on its capabilities; therefore, flexibility in processing a variety of constructs would be required.

(2) Portability between systems/host tools: the system should be retargetable to new hosts and environments, (i.e., new operating systems/DBMS's) with no major recoding necessary. The degree of retargetability would ultimately depend upon the initial system design and/or implementation; should it prove too system- or tool-dependent, then any future retargetability attempts would require considerable recoding to eliminate such interdependencies.

(3) Reduced complexity: the design should be made using a hierarchical methodology that encourages modularity, abstraction and independence. Thus, the complex task of processing NL queries would be decomposed into more manageable, simpler tasks that can be implemented independently. The integration procedure should also follow similar guidelines. The resulting design would then consist of a tree-like structure of modules, each performing a single task, with well-defined and uniform data exchange.

(4) Efficiency: as data retrieval is a process that requires "visible" man-machine interaction, response time is very

important. This is especially true for casual users who often do not realize the complexity of the retrieval process and expect "instant" response. Therefore, system processing time for translating NL to formal queries should be reduced to a minimum, necessitating a highly optimized design. In addition, resource usage such as disk accesses, main memory requirements, special I/O devices, etc., should also be minimized.

Using these objectives as guidelines for system design, the methodology of the system development process will be presented in the next section. The objectives presented here are general; more emphasis on NL related aspects, as well as the specific objectives, will be examined in the next section, where the NL related system design objectives will be presented and explained.

CHAPTER 3

THE HIGH LEVEL DESIGN OF KARL

3.1 INTRODUCTION

KARL is a software system designed for understanding restricted natural language within a retrieval environment. As such, it has design objectives which are related to natural language processing, as well as design objectives which are considered more general and applicable to any software system.

In this section, both classes of design objectives will be examined and the high level design of the system will be presented. In several aspects, KARL deviates from traditional natural language systems. These differences will be presented. In addition, the state-of-the-art will be presented in the design alternative areas addressing technology that is available for use in designing and implementing NLQS.

3.2 GENERIC OBJECTIVES REVISED

The generic objectives called for a number of desirable characteristics that the resulting system is intended to possess. KARL design focuses on a number of these. Specifically, the characteristics that KARL has, as dictated by the generic objectives, are as follows:

(1) Adaptability to new applications: one of the main problems in today's NLQS's is their inability to function within a variety of applications. KARL allows retargeting to different applications by allowing the user to redefine the Knowledge Base contents relevant to the application. Then, any application (within limits, of course) can be handled without modifying the programs themselves.

(2) Portability between systems/tools: KARL is implemented on the UNIX operating system and the Ingres relational DBMS [Epstein 79; Stonebroker 76]. It is expected that KARL can be ported to other operating systems with minor changes only. This is achieved by using only one system dependent call ("system()" for command level escape) that is typically available on most operating systems, coding all parts of KARL in the "C" language which is highly portable and available in a wide variety of operating systems and hardware configurations. This is also true for converting KARL to operate with different DBMS's, as its embedded query language constructs have been selected and structured based on calls available in many of today's modern DBMS's using embedded query

languages.

(3) Reduced complexity: Many successful natural language programs have been written in AI-specific languages, like LISP, PROLOG, etc. [Winston 81; Rich 83]. However, such languages, while convenient for development, are typically not suited for interface with "real" existing DBMS's. Also, the programming complexity increases due to the restrictive nature of such languages for general-type programming. KARL is written entirely in "C" [Kerningham 76]. Also, the underlying concepts of KARL, to be discussed in more detail in the next chapter, are relatively simple, thus yielding a less complex design than other NLQS's available [Wasserman 85; Salton 83].

(4) Efficiency: KARL is implemented using simple programming constructs and fixed memory configurations in order to avoid complex subroutine invocations and dynamic storage allocation and reclamation overhead. Execution efficiency is improved by using the efficient, optimized "C" compiler available on UNIX [Kerningham 76]. For further investigation, Chapter 5 presents metrics of KARL overhead in the retrieval process, using Ingres and UNIX. It should be noted, however, that KARL's prototype design and implementation necessitates an approach that emphasizes convenience and flexibility rather than

performance. Also, Chapter 5 discusses, as part of future research areas, production-level optimization techniques that can be used for further performance improvement.

3.3 SPECIFIC OBJECTIVES

Apart from the more general, software engineering criteria that essentially recommended the first set of generic objectives, there are several NLQS and DBMS related aspects that the design of KARL must handle. These aspects are as follows:

(1) Knowledge storage, processing, and acquisition capabilities that assist in system retargetability.

(2) Grammatical constructs handling capabilities that allow recognition of different forms of the same word. Also, capability of handling synonyms.

(3) Syntactic construct handling capabilities that allow recognition of different syntactic forms of questions.

(4) Semantic construct handling capabilities that allow verification of different semantic forms of questions.

(5) Learning capabilities that allow a system to "learn" new words and constructs.

(6) Handling of elliptic queries, thus necessitating heuristics in order to understand and process such queries. Also, capabilities for generalized error detection and appropriate reporting.

In the following six sub-sections, the NL specific objectives will be examined and the methodologies followed to provide solutions to these objectives will be presented and explained.

3.4 SYSTEM KNOWLEDGE

Knowledge is used to augment the process of natural language interpretation and assist in resolving ambiguities that might arise from the user's English input. Extensive research has been undertaken on the subject of knowledge, in particular, knowledge acquisition, representation and usage [Winograd 83; Taylor 84].

KARL's capabilities for learning, system and application independence, and relatively easy retargetability benefit from its ability to store, manipulate and retrieve knowledge stored in a machine-readable form. Thus, certain key functions of the Knowledge Base Management System (KBMS) can be viewed (and implemented) as DBMS operations. For example, knowledge addition would involve additions to the Knowledge Base (KB), while retargeting to a new application would involve re-populating the (relatively small in relationship to the DB) KB.

There are three aspects on which the KARL system is based with respect to using knowledge to process queries:

(1) Knowledge Acquisition is performed either at system initialization time or during actual use. Acquisition is highly dependent on the knowledge type; for example, knowledge of English language structure and syntax is not likely to be acquired at use time, whereas entity-specific knowledge can be initialized to an operational minimum and grow as a system is being used.

(2) Knowledge Representation involves storing the knowledge in a machine-readable form that can be used by the system. Knowledge representation is typically handled by the Knowledge Base Management System and is independent of the application [Wiederhold 84]. The approach that is to be followed in KARL will use the host DBMS to store knowledge. This approach eliminates the complexity and overhead of traditional KBMS's, since the knowledge required is relatively simple and does not involve complex interdependent representations. Knowledge about English syntax is "hard-wired" in the syntax analyzer, and can be extended by adding new patterns. Entity knowledge is stored in frames, which are defined as all

the information available for a particular type of entity and common for all instances of the same entity. The frame representation is altered to a table form and stored in the host DBMS. Relationships between entities and/or actions are encoded in function form similar to the first-order predicate calculus [Dahl 83], and are also transformed and stored in table form.

(3) Knowledge Utilization involves the use of knowledge in query processing [Wiederhold 84]. Syntactic analysis uses knowledge of allowable English question forms to syntactically verify an input sentence; semantic analysis uses knowledge of domain-specific terms, ranges, and relationships; pragmatic analysis uses more common knowledge to complete the semantic verification; database schema mapping uses schema knowledge to map input terms onto DBMS constructs; and query generation uses knowledge of the DBMS formal query mechanism to create the final query.

KARL's learning capability benefits from the presence of a redefinable KB. The user is able to redefine terms during a session and thus "teach" the system new terms and constructs. The main knowledge entity that is user-accessible is the Intelligent Dictionary (ID). Its purpose is to maintain the knowledge of words and multi-word sequences that are known for the application being used. The ID is implemented as a table collection. It contains entries for different word classes, namely:

(1) Nouns (database table and attribute names)

(2) Verbs (action & property indicators)

(3) Single-word Synonyms (replacing and eliminating query terms)

(4) Multi-word Synonyms (replacing query terms)

(5) Adjectives (property and condition indicators)

In addition, the ID contains a list of all known words irrespective of class, to assist identification and improve efficiency by performing a two-step access. Finally, the ID contains a frame collection of all entities in the database and their associated characteristics. The DBMS that implements the Knowledge Storage and Retrieval System is responsible for the encoding and storage of the ID.

3.5 GRAMMATICAL CONSTRUCTS HANDLING

Grammatical constructs handling involves the addition of English language rules and methods for determining the "stem" of words from alternate forms, so that they can be compared against the contents of the ID and positively identified (or diagnosed as such, if unknown). Such transformations generate one stem or general term from singular or plural forms (for nouns/adjectives) or past, present or future forms for verbs. For example, Figure 6 indicates several forms of entries that are handled through grammatical transformations.

```
student      earn         salary       made
student's    earned       salaries     make
students     earning                   will make
   |         will earn       |            |
   |            |            |            |
   V            V            V            V
student        earn       salary        make
```

Figure 6 Grammatical Transformations

In addition, the grammar transformation of the input query involves handling of noisewords such as articles, meaningless words, punctuation, etc., and recognition/classification of certain word types such as literals and numeric constants. Also, replacement of synonyms occurs during this phase. The result is a "cleaned" query that is ready to be passed to the next processes.

In order to determine the grammatical structure of the input query terms, the most common rules of the English language are implemented through a rule base. Since this form of knowledge is "stable", i.e., does not change with each application, the knowledge can be programmed directly into the grammatical transformation and recognition component of the system, and subsequent updates that may be needed can be implemented by additions/modifications of the system source code. As stated earlier, however, the rules of the English language do not change, so this method is viable.

Irregular words and constructs are stored as synonyms. If all rules for the recognition of a term fail, then it may be a synonym. For example, "made" (the past of "make") is stored as a synonym and direct replacement of "make".

3.6 SYNTACTIC PATTERN RECOGNITION

Syntactic pattern recognition and subsequent verification is the phase of the query processing cycle at which the query is verified according to accepted syntactic rules of the English language (or restricted subset thereof). Syntactic recognition and verification is very important in the query processing cycle since it allows the query processor to determine the syntactic structure of the input query and take subsequent actions based on this structure [Jones 83]. Also, it allows early error recognition and even suggestions to the user.

Until recently, syntactic verification was the only means of verifying a natural language query. That is, syntactic verification was conceived to be adequate for accepting a NL query as valid. This methodology was shown in early general-purpose dialog systems like ELIZA and PARRY

[Winograd 83], and later in NL DBMS interfaces such as PLANES and INTELLECT [Wassermann 85]. While it is true today that syntactic verification alone is not adequate proof of correctness, still, it comprises a significant amount of the entire query processing cycle and therefore deserves special attention.

There exist methodologies for sufficiently correct recognition of English grammar [Salton 83]. Mapping these techniques to the subset of the English language that constitutes questions and answers provides sufficient syntactic verification capabilities [Leslhart 76]. Salton [Salton 83] identifies three main types of syntactic analysis frameworks:

(1) Phrase structure grammars, that map most of the language properties into structured elements.

(2) Transformation grammars, that analyze distinct subsets of the sentence into equivalent fragments based on transformation principles.

(3) Network grammars, that construct a network from the input sentence and apply rules to its structure.

Transformation and network grammars are almost equivalent in the sense that both can be used to represent the English language constructs sought. KARL uses the network grammar approach. Due to simplicity considerations, the recursive transition network is used instead of the augmented transition network [Tennant 81; Winograd 81]. A sample query and its associated recursive transition network (RTN) are shown in Figure 7.

```
Show the good female students enrolled in "CMPS150"
         |          |         |        |
         V          V         V        V
   show good female student enroll "CMPS150"

         verb          noun         verb       literal

->  a  ------->  b  ------->  c  ------>  d  ------>  e

                / \           |           |
                \_/           \___________/

              adjective          noun
```

Figure 7 Recursive Transition Network (RTN)

As was the case with the grammatical constructs that remain unchanged over different applications and DB contents, the knowledge represented in the RTN is considered "stable" and therefore is suitable for implementing directly in the program source code. Implementation details of the RTN structure can be found in Section 4.5.

The effect of the syntactic analysis and verification is full identification of the sentence structure, association of the input sentence structure with known (correct as well as incorrect) sentence pattern structures, and further clarification of ambiguous terms that were not properly resolved by grammar rules alone. For example, the word "love" is both a verb and a noun, thus necessitating delayed identification. The result of the syntactic analysis module is a list of terms (tokens) that are fully identified within the scope of the ID. This list is then passed for subsequent analysis to the semantic verification module.

It should be noted that the syntactic and semantic verification modules do not interact with each other. Several research methodologies suggest a more integrated approach that integrates syntactic and semantic analysis. Such approaches, however, are more practical in the solution of the general natural language understanding problem and are too complex for a subset-based application such as NLQS.

3.7 SEMANTIC VERIFICATION

A NL query is finally verified as correct (and thus acceptable) by the NLQS if its semantics are correct. Semantics can be widely defined as the aspects of the query that refer to the meaning of entities, regardless of grammar and syntax [Kalz 72]. This phase of the query processing cycle is important since it is the last step in the flow of the NL query within the system. Casual users are more prone to make it difficult to detect semantic errors than experienced users. In general, as is the case with programming languages [Pratt 83], semantic errors are more difficult to detect than syntactic errors and suggest positive measures for correction.

Semantic verification is performed on two levels: the linguistic level and the database schema level. The linguistic level handles verification based only on linguistic semantic criteria, while the database schema level handles verifications based only on database schema-related criteria. The rules that are used for linguistic semantic verification are traditional English semantic-based rules, while the database schema itself, (actually an expanded view stored in frames) provides the database-related verification rules. In both cases, the two sub-procedures are distinct with no interaction due to functionality considerations.

Different criteria apply for the two sub-processes; similarly, there are different knowledge requirements involved. The two sub-processes with their associated knowledge requirements are addressed in the following two sub-sections.

3.7.1 DATABASE RELATED VERIFICATION

Database related verification involves checking the input query for semantic inconsistencies against the database schema, or an expanded version that includes semantic constraints [Brodie 84]. Therefore, the queries that are syntactically correct but semantically incorrect (in relationship to the DB itself) can be detected. Several types of inconsistencies are handled by KARL:

(1) Invalid attribute names and table names.

(2) Concordance of attribute and table names.

(3) Values of literals out of range.

(4) Incorrect literal patterns.

(5) Inappropriate operators associated with operands.

System knowledge of the entities present, represented by case frames in the knowledge base, is used to detect the inconsistencies. Figure 8 shows examples of such errors, numbered as the causes above:

```
(1) display manager for city "Detroit"
        (No "manager" in the database)

(2) display prices of cities where name = "Dallas"
        (City relationship does not have price)

(3) print flights with prices less than "$0.25"
        (there is a low limit on all flight prices)

(4) print city with code equal "AX123&"
        (invalid literal, all codes are 3 chars long)

(5) print flights with name greater than "LFT"
        (operator "greater than" can not be applied)
```

Figure 8 Database Related Semantic Errors

The process of database related semantic verification is able to detect and indicate different classes of semantic errors that the user does not realize. The output of this verification is then forwarded for linguistic verification.

3.7.2 LINGUISTIC RELATED VERIFICATION

Linguistic verification involves checking the input query against a set of linguistic-based rules of correctness. There may be the case that syntactically the query is correct (that is, using English syntax as the criterion), but the combination of words produces an incorrect meaning.

Linguistic verification involves knowledge of the interrelationships of words based solely on meaning [Lehnert 76]. Thus, the linguistic knowledge is dynamic and needs update capabilities, so it is implemented as a part of the user-defined knowledge base.

The verification process involves checking the possibly incorrect constructs against the contents of the appropriate entries in the knowledge base. KARL supports several such checks, including:

(1) Concordance of noun-noun constructs.

(2) Concordance of noun-verb constructs.

(3) Agreement of noun-adjective combinations.

Linguistic verification is considerably more complex than database related verification. This is because the knowledge encoded in the database schema and its frame expansion is relatively simple to verify, while this is not the case with a more complex linguistic semantics problem. Also, linguistic verification depends on the human meaning that is associated with words and constructs, which is not always simple to convert to a machine-readable and processible form. Examples of linguistic semantic inconsistencies can be seen in Figure 9.

(1) who is taking a rich course?
("rich" and "course" don't match)

(2) print the courses that earn "$13,000" a year
("earn" is associated with "faculty" or "student")

(3) What is the salary of a good car?
("car" does not include a "salary" attribute)

Figure 9 Linguistic Semantic Inconsistencies

3.8 LEARNING CAPABILITIES

Learning is a capability that has long been associated with humans and animals only. It is a process that involves acquisition of certain elements encountered in a task and later utilization of these elements. Knowledge is distinct from data in that it does not change as dynamically as data, but in many cases remains relatively stable.

Knowledge capabilities, however, dictate that the system (or human) must have the ability for acquiring, transforming (if needed), storing and later retrieving and using knowledge for use in a given task. In a NL processing system, this capability is crucial if retargeting to a different application is being sought, or redefinitions/updates are performed on the database schema and overall organization. KARL uses learning to its benefit in a number of different areas:

(1) Learning is used to aid retargetability to different applications. The entire re-initialization can be performed through a massive learning process, or read through prepared file(s).

(2) Knowledge updates for a variety of reasons (performance improvement, debugging the KB, etc.) are convenient.

(3) There-exist capabilities for incorporating new terms as either entities or relationships between entities, in the form of nouns and adjectives or verbs, respectively.

(4) If the system encounters an unknown term, KARL is able to interactively ask the user for the type of term, and

its known properties. A simple fill-in-the-blank form is used.

The last characteristic has been very important as it allows the users to customize the knowledge base. For a prototype model such as the one presented in this thesis, no security constraints have been considered. A production environment may set update/append restrictions which can be implemented through the relational database system underlying KARL, using its security system.

3.9 ELLIPSIS AND AMBIGUITY HANDLING

Ellipsis and ambiguity are present in many forms of human-to-human communications [Kalz 76]. However, while they can be tolerated and understood by humans, a system is typically not able to understand and process such sentences.

Ellipsis is a form of speech in which certain parts of the sentence structure are omitted. The purpose of ellipsis can be either as a figure of speech or for convenience. Typical forms of ellipsis include pronoun reference, missing noun phrases or missing verbs. KARL has not incorporated pronoun references, although a framework for inclusion is presented in the conclusions section. It can then be seen that simple pronoun reference implementation involves backtracking and maintenance of query histories which can be included in KARL at a later time.

As indicated earlier, several forms of ellipsis handling are provided within KARL. These ellipsis forms are found in typical English phrase structures. These forms include:

(1) Missing nouns, where a noun is either a relation name or an attribute name. Then, context analysis is required to determine the missing term(s) and incorporate them into the intermediate query.

(2) Missing operators in the case of conditional or relational statements. The default values are determined by consulting the appropriate frames in the Knowledge Base.

(3) Missing verbs. If an action verb is missing, then "select" is chosen by default. If a relationship-indicating verb is missing, the context system is used to insert the appropriate term.

Ellipsis is typically handled in the syntactic analysis and verification module, by including elliptic sentence construct patterns in the database of patterns and transforming them into non-elliptic structures for further processing. Figure 10 displays ellipsis handling in KARL.

```
(1) who is rich?
         (ascertains that "rich" is used with "salary",
          which in turn is used only with "faculty")

(2) who is "John Doe" ?
         (ascertains that "John Doe" is a free-form
          string, address or name, but since it has
          "who" it is a name within student or faculty)

(3) print names of students in "CMPS150"
         (ascertains that "CMPS150" is a course
          and use the proper form to complete the query)
```

Figure 10 Ellipsis Handling Capabilities in KARL.

Ambiguity is also a common feature of the human's process of speech. Ambiguity may arise in a NL statement when a query interpretation process attempts to associate more than one meaning (or term interpretation) to the same term. In order to fully process the query, the NL interpreter has to decide on only one meaning which will then be bound with the term. If the NL processor is not able to determine the exact meaning, then either heuristics have to be applied, the user queried for additional explanation or the query process is abandoned.

Ambiguity in KARL arises when words which can be of multiple type definitions in the dictionary are used, or when a qualifier in a multiple word construct is omitted. Most ambiguity is considered linguistic ambiguity, and the heuristics applied attempt to clarify the construct by applying semantic information provided in the KB.

KARL implements a solution by trying to eliminate to the highest degree the ambiguity that exists as part of the KB definition. If there is ambiguity present, then certain heuristics will be applied and if the heuristics also fail, the order of the entries determines the default. Therefore, the terms of an ambiguous entry in the KB are arranged in a likelihood order, thus assisting the selection. If this also fails, then the user is presented with the sequence of possible interpretations and requested to select one. A sample session in which ambiguity arises and the user is queried is presented within the KARL sample session contained in Appendix B.

3.10 AN OVERVIEW OF THE QUERY PROCESS CYCLE

A NL query processing system can be considered as a never-ending design process, since new features, originating in the English (or human) language are considered for inclusion and eventually included in the design. Therefore, there exists a line between implementability of the design and lack of features that limit the NL processor's performance.

Aside from the details of implementation, many natural languages have common structures. This can be compared to compilers, where almost every compiler has common processes with other_similar ones, i.e., token generation, lexical analysis, grammar analysis, code generation, optimization, etc., and also components commonly used, i.e., the parser, code optimizer, etc. [Aho 79]. Since KARL and mostly any question-answering system can be thought of as a type of compiler, the same methodology of basic components and features is followed. The query processing cycle and common requirements for the understanding of NL queries is presented next.

Natural language understanding in general involves at least three distinct procedures that may be independent of each other. The three procedures are known as Syntactic, Semantic and Pragmatic Analysis [Salton 83; Blanning 84; Winograd 83]. These steps are typically sufficient for general-purpose natural language understanding applications, but additional steps are required in order to process database queries, i.e., questions. The additional tasks performed by a database front-end should also include schema mapping and formal query generation in order to provide the capabilities needed for the query translation process.

3.10.1 LEXICAL ANALYSIS

Lexical analysis involves recognition of the individual terms of the query and generation of the intermediate form that is used to represent the NL query throughout the program. This phase varies from one system to another. Query "clean-up" and grammar operations also occur on this level.

This step of the query processing cycle is often integrated within the terminal monitor/user interface. Although not many systems have a grammatical processor, it is of high value since the number of words stored in the dictionary is drastically reduced. In addition, features such as spelling checking can be incorporated and even switched on/off, without further implications.

3.10.2 SYNTACTIC ANALYSIS

Syntax refers to the relative position of words and word sequences in a sentence, taking into consideration syntactic restrictions only [Markus 82]. Syntactic analysis is necessary to determine the structural correctness of the sentence.

NL syntactic analysis can be presented in a way similar to the syntactic analysis of computer languages. The meaning of entities is not involved in the process. Syntax rules are used to determine the correctness (or acceptability) of the user's NL input. Also, there must be provision for "pidgin English" (i.e., semi-formal query) handling, since users may be using such input.

3.10.3 SEMANTIC ANALYSIS

Semantics refers to the meaning of words and relationships associated with application-dependent terms/words in sentences [Charniak 76]. Pragmatic analysis, a part of semantic analysis, attempts to further semantically verify the correctness of the input sentence by using general, application independent concepts [Salton 83].

Not many existing programs perform a per se semantic analysis. Many ATN-based systems perform syntactic and semantic analysis at the same time, using the ATN's network structure for syntactic checking and the register contents of the ATN for the semantic conditions that must hold. This produces a method of semantic verification [Bolc 83]. However, semantic verification at the abstract level is a task that is considered separate from syntactic verification.

3.10.4 FORMAL QUERY GENERATION

The next step in the natural language query process is the translation of the query into the formal query syntactic and semantic constructs. This process often involves compiler-related manipulations, such as code generation and possibly optimization [Aho 78; Hunter 81].

Depending on the system capabilities, the code generation can be retargetable to different hosts, i.e., generate formal queries that are suited for execution on different systems. Such capabilities have the potential for multiple database and information system usage [Hall 85], and are highly desirable.

3.10.5 FORMAL QUERY EVALUATION

The final step in the NL query processing cycle is the evaluation of the formal query generated by the NL processor. This is performed by either generating the appropriate high-level formal query and passing it to the DBMS for interpretation and execution, or by decomposing the formal query generated and invoking the low-level DBMS routines in order to execute it. The choice would be made depending on the facilities that the host DBMS provides. The results are then displayed.

3.11 KNOWLEDGE AND QUERY PROCESSING

With knowledge of English language terms and constructs, the role of knowledge in the query processing cycle becomes extremely important. The phases presented above all assume certain knowledge types to be available in order to assist the query processing cycle.

Knowledge is divided into two types: the knowledge that is required for the application, and general knowledge that is required to process any query that the system can handle. The first type of knowledge is called "dynamic" knowledge in KARL, since it tends to change with time (i.e., knowledge base improvement or learning), or with the application (retargeting). The second type of knowledge is considered as "static", and is based on general principles applicable to question-answering. Such examples of static knowledge include knowledge of suffix-removal rules, grammar rules that determine the appropriate form of sentences and sentence fragments, and the syntax of a target formal language into which the NL input is translated by cycling through the processing cycle. Figure 11 displays the cycle, as well as the knowledge required. The knowledge is tagged as either static or dynamic by the marker (s) for static and (d) for dynamic.

```
     Input
     Query
      | |
      \/
+------------+
| LEXICAL    |      Intelligent Dictionary        (d)
| ANALYSIS   |      Grammar Knowledge             (s)
+-----++-----+
      | |
      \/
+------------+
| SYNTAX     |      Syntax Knowledge              (s)
| VERIFIER   |      Schema Knowledge              (d)
+-----++-----+
      | |
      \/
+------------+
| SEMANTIC   |      Semantic Knowledge            (d)
| VERIFIER   |      Schema Knowledge              (d)
+-----++-----+
      | |
      \/
+------------+
| FORMAL     |      Schema Knowledge              (d)
| QUERY      |      Formal Syntax Knowledge       (s)
| GENERATION |      Formal Semantic Knowledge     (s)
+-----++-----+
      | |
      \/
+------------+
| FORMAL     |      Formal Syntax Knowledge       (s)
| QUERY      |      Formal Semantic Knowledge     (s)
| EVALUATION |      DBMS Specific Knowledge       (s)
+------------+
```

Figure 11   The NL Query Processing Cycle

3.12 HIGH-LEVEL DESIGN OVERVIEW

Although the process of understanding general human input has been too complicated for machines to perceive with an acceptable degree of comprehension, special-purpose understanding programs such as abstracting, indexing or NL query systems have been able to function properly, sometimes even to production-level quality. Careful system design, that does not attempt to be a "one-in-all" type of solution, but rather focuses on the problem that is to be solved, is the answer.

Software design techniques such as functional decomposition and abstraction allow separation of tasks and creation of what is essentially an "airtight" processing system with highly individualized functions [Warnier 79; Freeman 81]. Although the task of comprehending NL queries is difficult by any means, decomposing the problem into small subsets, for which there are often answers (i.e., lexical and grammar analysis, formal query generation, etc.) is a method that can be more practical to design and implement than the highly complex approaches so far.

Following the divide-and-conquer approach [Aho 79], the real design problem is not forming the solution but rather defining the problem in terms such that a computer solvable approach is viable. Once the individual problems have been identified, a uniform representation form for the information that is communicated between modules is required. Once a module is defined, the internal transformations that are performed on the input query must be localized to avoid their propagation throughout the entire system.

Concluding the high-level design of the KARL system, the important techniques and concepts introduced in this chapter will be applied in the next chapter which discusses the low-level design and implementation process. Such techniques and concepts include task separation amongst modules, high functionality, independence and simplicity.

CHAPTER 4

LOW-LEVEL DESIGN AND IMPLEMENTATION

4.1 THE PROTOTYPE FRAMEWORK

The principal concepts that KARL is based on are simplicity and use of modern software design techniques to obtain both implementation capability (i.e., have a design that is implementable) and NL handling capabilities that can be used for query processing. These concepts have not been used extensively in other NL query processing systems [Wasserman 85; Taylor 84] and the results can be seen as systems that are not flexible in handling queries [Taylor 84; Blanning 84] (low NL handling capabilities) or difficult to implement and maintain [Weizenbaum 66; Wasserman 85].

To prove the validity of the design concepts used in KARL, an experimental computer program was developed. The design and implementation of the prototype, the KARL 1.02 system, is presented in this section. The prototype is implemented on a Digital VAX-11/780 computer running the UNIX operating system, Berkeley 4.2 distribution [Kerningham 79]. The entire prototype is implemented in the "C" programming language.

KARL interfaces with Relational Technology's INGRES relational database system, Version 7. The interface is possible through system subroutine calls to the DBMS monitor for the query processing, and embedded DBMS code contained in the knowledge processing routines. In addition, the "LEX" tool for generating regular expression recognizers is used, as it accepts regular expressions and generates finite-state automata that recognize them. "LEX" generates portable "C" code [Lesk 76].

4.2 LOW-LEVEL DESIGN METHODOLOGY

Because proven techniques from compiler construction and traditional software design methodologies were used, as outlined in the last part of Chapter 3, the low-level design and implementation of the KARL software system is simple to understand. Compiler techniques such as regular expression recognition, lexical analysis and intermediate code generation are used in the implementation of KARL [Aho 79; Hunter 81]. More general principles such as modularity, top-down design and functional decomposition are also used.

KARL is knowledge-assisted, using knowledge to assist the retrieval process. Knowledge is represented in machine-readable form and stored in the KB. Then it is used to assist the translation process of the input NL query. The issue of representation of real entities as abstractions handled by the software is the main aspect that KARL benefits from. Data representation, therefore, is the main issue of the KARL low-level design. Input consists of both knowledge, either contained in the program structure or encoded and stored in the KB, and the NL query, as stored by the NLQS monitor system.

Manipulation of the NL query with sequences of transformations, from the NL query, to the formal query, is the main process that occurs within KARL. No specific intermediate query representation is used except the original data structure (list of words and types) that is initialized after the NL query is read in and manipulated as each module performs its transformations to it. Thus, the orthogonal design methodology is followed with no exceptions.

In order to obtain the appropriate transformations, each module of the system performs an independent task. A top-down organization of the operations that are performed on the data structure that holds the query is used. Each operation occurs in a defined location within the entire process, with no interdependencies of either data or operations. The "black box" approach in the design methodology has several advantages over highly interdependent method-specific internal representations [Sommervile 82]:

(1) Convenience of additions/updates to the techniques used in the system. By avoiding dependencies of the entire program on certain segments of code and making all segments operate based on one input and one output, new features can be added by literally "plugging in" modules in the appropriate locations.

(2) Design efficiency. This is the result of the designers being able to concentrate on one problem only, with no concern for side-effects. Since little interdependency exists amongst modules, this approach is feasible.

(3) Error isolation and improvement considerations. Should a module malfunction due to design and/or implementation errors, a different design can be tested with few constraints. This is also true in the performance improvement issue, where the designer can determine defects and improve any malfunctioning modules with no effect on properly operating modules.

Abstract software design methodology is coupled with the generic and specific objectives presented earlier in order to provide the framework for the implementation of KARL 1.02. In this chapter, the details of the low-level design and implementation, in essence the internals of KARL, will be presented.

4.3 DATA STRUCTURES

Data structures are the logical structures in which information is stored. KARL uses data structures to store the NL query as it is being transformed into a formal query, and also to store application-dependent components of the KB.

In selecting the data structures to be used, considerations regarding programming languages, applications, and complexity have to be made. If the design of the data structures has a flaw, then the flaw is propagated as the data structure is used in the program. Also, if the data structure is complex, the possibility of side effects increases. Finally, the representation has to be simple, in order to conform with the framework of the implementation. There are two major concepts represented; one is the NL query itself and the other is the application-dependent, dynamic knowledge.

4.3.1 QUERY REPRESENTATION

There have been several "traditional" data representation schemas for the internal storage of database queries. Network models have been popular, in simple as well as complicated (i.e., augmented) forms. KARL uses a simple linear structure that consists of two lists. The first list is the list of tokens and the second is the list of token type identifiers. The structure can be seen in Figure 12:

```
+-----+----------+         +-----+------+
| NO. |  token   |         | NO. | type |
+-----+----------+         +-----+------+
      |                          |
      V                          V
+-----+----------+         +-----+------+
| NO. |  token   |         | NO. | type |
+-----+----------+         +-----+------+
      |                          |
      V                          V
+-----+----------+         +-----+------+
| NO. |  token   |         | NO. | type |
+-----+----------+         +-----+------+
      |                          |
      V                          V
  .  .  .  .  .              .  .  .  .
  .  .  .  .  .              .  .  .  .
  .  .  .  .  .              .  .  .  .
  .  .  .  .  .              .  .  .  .
```

Figure 12 Structure of NL Query Storage Area.

The representation contains sufficient information so that the various knowledge processing elements can identify the token as being of certain types and perform the actions required. As different parts of the system use different areas of the knowledge base, inefficiencies in this schema are reduced to a minimum (i.e., retrieving the same data more than once). A sample query can be seen in Figure 13.

$$\begin{tikzcd}
\text{FORMAL QUERY} & & \text{FORMAL QUERY (with no noisewords)} & \text{TOKEN PATTERN} \\
\text{print} \arrow[d] & & \text{print} \arrow[d] & \text{Verb} \arrow[d] \\
\text{all} \arrow[d] & & \text{student} \arrow[d] & \text{Noun} \arrow[d] \\
\text{students} \arrow[d] & & \text{enroll} \arrow[d] & \text{Verb} \arrow[d] \\
\text{taking} \arrow[d] & & \text{"CMPS351"} \arrow[d] & \text{Literal} \arrow[d] \\
\text{"CMPS351"} \arrow[d] & \Rightarrow & \text{\&} \arrow[d] & \text{Boolean} \arrow[d] \\
\text{and} \arrow[d] & & \text{live} \arrow[d] & \text{Verb} \arrow[d] \\
\text{living} \arrow[d] & & \text{"Lafayette"} & \text{Literal} \\
\text{in} \arrow[d] & & & \\
\text{"Lafayette"} & & &
\end{tikzcd}$$

Figure 13 A Sample Query and its Representation

4.3.2 KNOWLEDGE REPRESENTATION

Data structures for knowledge representation refer to the storage techniques of the dynamic parts of the knowledge base. As the dynamic part is required to change with the applications, there is a need for the ability of storing, retrieving and updating such knowledge.

The solution presented in KARL is to use the host DBMS's facilities of defining and handling tables (relations) for storing the dynamic parts of the knowledge base. Although there is a performance penalty for such a solution, the ability for rapid prototyping as well as the handling of changes that come as the design evolves overshadows the efficiency penalty. Should efficiency become a higher priority, such as may be required in a production system, a more efficient solution based on a memory-resident table driven KBMS can be implemented while maintaining the operational compatibility with the rest of the software system.

Dynamic knowledge is represented as a collection of tables. The table collection is implemented through a relational database system schema. The storage representation for the dynamic components of the KB is presented in Figure 14. A sample knowledge base for the university database that is used throughout the example is presented in Appendix A. The contents of the dynamic database were empirically determined, using basic database theory and linguistic/grammar references regarding the rules of the English language that handle the words present in the knowledge base. Learning capabilities also assisted the knowledge base building process.

Database Related Knowledge:

Noun Frame

Name	Type	Datatype	Max	Min	Pattern	Unit

Linguistics Related Knowledge:

Synonyms Representation

term	stands for

Verbs Representation

verb	subject	object

Adjective Representation

Adjective	Noun	Implied_property

Dictionary Representation

Word	Word_type

Multiword Representation

Term	Pattern_id	Rank

Figure 14 Dynamic Knowledge Representation Schema

Figure 14 presents the entire schema of the dynamic part of the KB. The individual tables represent the following knowledge:

(1) Noun frame: contains the knowledge that is required for the nouns part of the vocabulary, which are taken as either attributes of tables or table names. The knowledge contained is the noun name, type, data type (i.e., real, integer, string), its maximum and minimum values if appropriate, an optional pattern that is required in its literals, relation name in which it belongs, and its allowable operations (comparison, aggregation, etc.). Most of the knowledge is used for the semantic verification of the input query.

(2) Synonym representation: represents word pairs that are considered synonyms for query processing requirements. Noise words are contained as synonyms to the empty or null string. The table contains the term and the term it stands for.

(3) Verbs representation: verbs are associated with subjects and objects and the verb section of the dynamic knowledge base contains such knowledge. Specifically, for each verb, a noun is associated as subject and another as object. This holds true for retrieval

purposes only, and verbs with both direct and indirect subjects are not considered, i.e., "A student earns a grade" may be used in a query but "the teacher threw him the ball" type of construct with direct and indirect noun objects are not handled, as it is far less frequent in retrieval contexts than subject-verb-direct object questions [Lehnert 78]. This table is used for semantic verification as well as ellipsis or plethora handling, in cases where the verb is supplied but not its indirect object (ellipsis) or in cases where both are specified and one has to be rejected (plethora).

(4) Adjective representation: adjectives are associated with properties (similar to property lists in LISP [Winston 81]), but only when associated with certain nouns. So, "good student" would imply a student whose GPA is more than a certain amount, but, at the same time, "rich car", although syntactically correct (as a noun phrase), is semantically incorrect. Adjectives are used like verbs, and also during the formal query generation phase where adjectives are replaced by their property.

(5) Dictionary representation: all words known to the system are contained in the dictionary, including punctuation and noise words. If an unknown word is found, a number of grammar and lexical rules will be applied and, if these fail, the user will be queried. This table is

consulted only during lexical analysis.

(6) Multiword representation: this table contains entries for noun sequences, which are traditionally very difficult to interpret otherwise. Such sequences are "social security number", "home address", etc. The patterns are considered as synonyms to single-word terms, i.e., the sequence "social security number" yields "ssn" which can be identified as a noun in the dictionary and the noun frame lists.

4.4 LEXICAL AND GRAMMAR ANALYSIS

The lexical analysis phase of the compiler is typically defined as:

> "The phase of the compilation that separates characters of the source language into groups that logically belong together; these groups are called tokens. The tokens are keywords, identifiers, operand symbols and punctuation. The output of the lexical analysis phase is a sequence of tokens, the token list."

This definition was introduced by [Aho 79]. KARL, utilizing a number of compiler construction techniques, uses lexical analysis in order to separate the tokens, combine them where applicable, identify the tokens as terms being either literals, operands or operators, and generate the token list that is used as the the next phase input.

As the English language permits transformations of the terms in the form of tenses or clauses, the system can perform heuristic tests and apply English language rules in order to determine the word type and identify the word. Should different forms of the word be used, then the program can perform the appropriate combination of transformations and determine the word type.

The grammar analyzer can detect the appropriate stem and identify the word. This approach results in relatively more complex code than maintaining the list of all combinations of word forms in the dictionary. However, dictionary size is drastically reduced as only the basic word (stem) is needed, thus yielding one entry per word. The exception of abnormal nouns and verbs is handled through synonyms.

Grammar and lexical analyses result in the initial stream of tokens and, where applicable, token identifiers. The sequence of tokens is free of synonyms, multiple sequence patterns and noise words. If there are still terms that are unknown although all rules have been exhausted, then the program queries the user to either correct the error (if any), replace the term with one that is known to the system, or redefine the term entirely. The system can then "learn" the new term.

The learning subsystem is invoked at the lexical stage, because most of the lack of knowledge is realized as the program is trying to process queries with unknown words. The user will be put in the knowledge redefinition subsystem, and then queried with the type of word that he wishes to select. Then, the user responds via multiple-choice type responses and defines the word as being known.

The process of lexical and grammar analysis is rather time-consuming, as it involves numerous accesses to the knowledge base and uses a potentially large number of heuristics in order to determine the word types, replace multiple noun sequences and eliminate noisewords. The process can be thought of as two independent sub-processes, namely lexical and grammar analysis. These processes are diagramatically shown in Figure 15 and Figure 16.

```
+------------------+
| Read NL Query    |
+------------------+
          |
+---------v--------+
| Replace Multiple |
| Sequence Patterns|
+------------------+
          |
+---------v--------+
| Generate Tokens  |
+------------------+
          |
+---------v--------+
| Replace Synonyms |
| and Noisewords   |
+------------------+
```

Figure 15 Lexical Analysis of Input Query

The process of lexical analysis will perform the following tasks:

(1) Read in the query, and determine query type (query, quit, help).

(2) Replace all multiword sequences with the appropriate nouns, so that only single terms occur (except literals).

(3) Generate the initial token list. All tokens are single words, with the exception of the literals which are enclosed in quotes and can contain blanks.

(4) Identify and replace all synonyms; also handle all noise words by eliminating them.

At this stage, punctuation has also been removed with the exception of symbols such as ">", "<=", etc. Then, the grammatical processing can be performed. A graphical presentation of the process for a single token is illustrated in Figure 16.

```
                     token
+----------+           |         +----------------+--------+
|          |           |         |                |        |
|       ___V___________V_________V_      +------+------+   |
|      /   Is Word in            \  Y   | Get next    |   |
|     <    Dictionary ?           >------>|  Token      |   |
|      \__________________________/     +-------------+   |
|                      |                                   |
|                      | N                                 |
|         _____________V______________                     |
|        /    Is Counter at          \  Y  +------------+   |
|       <   End Of Rules Yet ?         >---->| Query User |--+
|        \____________________________/     +------------+
|                      |
|                      | N
|            +---------V--------+
|            | Apply Next Rule  |
|            +------------------+
|______________________|
```

Figure 16   Grammatical NL Query Processing

The grammatical processing results in a string of tokens which have all been identified in the dictionary, or in querying the user for terms which are unknown. The process is as follows:

(1) If the word is in the dictionary, then identify the word and attach its token identifier.

(2) If it is not, then apply all known grammatical rules for suffix and prefix removal and replacement with proper forms.

(3) For each transformation, attempt to identify the word. If it is identified, proceed with the next one at step 1.

(4) If not identified, query the user and then either accept a replacement that is in the dictionary, learn a new term, or abort the query.

The rule base for the suffix removal contains 17 rules of modern English that convert tenses and voices. The rules are part of the static knowledge base, as they do not change with different applications. Should the user need to increase the scope of the rule base, the source code would need modifications. The structure of the program is explanatory and there are provisions in the source code for future updates (i.e., very few "hardcoding" constructs are included).

The outcome of the lexical and grammar analysis phases is the token list and the token identifier list. Both lists are passed to the syntax verifier for syntactic verification of the input query. This phase is described in the next section.

4.5 SYNTACTIC ANALYSIS AND VERIFICATION

Syntax verification has been the traditional method of determining the correctness of NL queries, with little concern being placed on semantics [Winograd 83]. Even with the shift towards more semantic analysis in the processing of NL queries, syntax verification and syntax-based NL systems are still popular [Wasserman 85; Tennant 81; Winograd 83].

Syntactic analysis in a programming language involves reading in the token sequences from the output of the lexical analyzer and verifying that the patterns occuring in the input are accepted in the language specifications. Often, the input sentence(s) are transformed into a tree-like structure called the parse tree [Hunter 81]. All subsequent operations on the sentence are performed on the (more structured) tree.

Since the NL processor accepts a subset of the English language that has a grammar and a syntax with rules, transformations similar to the ones performed by programming language compilers can be applied in order to verify the syntactic structure of the input sentences. Often, as is the case with programming languages, a parse tree (for transformational grammars) or augmented transition grammar (for ATN-based programs) is used. A number of successful NL query programs use either context-free grammars or network based grammars which perform extensive transformations to the input query.

KARL, being oriented towards more semantic-based query analysis, uses a significantly simpler mechanism for verifying NL or near-NL queries. The mechanism is based on simple recursive transition network grammars, simulated by regular expressions [Grimes 75]. As there is no specific intermediate representation (i.e., an ATN), the entire cycle is simplified. Simplification of the syntactic analysis phase results in simplification of the entire query processing cycle.

The method that is followed in KARL is derived from finite state automata based mechanisms. A finite state automaton recognizes inputs known as regular expressions [Hunter 81]. The regular expression constitutes a sequence of token identifiers that are bound together. If the regular expression is recognized as being acceptable for further processing, then the pattern family number is returned. Else, a syntactic error occurs. Within KARL, a regular expression is used to simulate the recursive transition network.

The finite state automaton is designed to recognize regular expressions. A regular expression is a string of characters (or symbols), from a given alphabet, combined under the rules of sequence, alternation, multiple occurrences, and grouping in logical sub-patterns [Hunter 81]. Since the input sentence is a list (string) of token types and identifiers, verifying the syntactic correctness of the query involves generating the RTN-based regular expression, passing it to the finite state automaton, and then receiving an answer from the automaton regarding the status of the input string. If the regular expression is accepted by the automaton, it can be concluded that the input sentence is acceptable syntactically.

Two important concepts must be presented before the entire semantic verification cycle can be explained. One is the individual lexical token identifier types (i.e., verbs, nouns, adjectives, etc), and the combinations of such token identifiers that are acceptable and allowed by the automaton. The implementation of the automaton through a regular expression recognizer generator is presented also.

4.5.1 TOKEN IDENTIFIER TYPES

The token types are the types that identify the grammatical classification of the input tokens. The token types are derived from English language word types. The token identifiers are as follows:

(1) Noun: a noun can be either a relation name or a relation attribute name. Symbol: "n".

(2) Adjective: an adjective implies a property to the attached noun in the noun phrase. Symbol: "a".

(3) Verb: a verb implies either action or relationship. Symbol: "v".

(4) Literal: a literal is the value specified by the user in a conditional retrieval. Symbol: "l".

(5) Boolean operator: connects various parts of the query, like "and", "or", etc. Symbol: "b".

(6) Relational operators: connect the noun with its associated literal, like "greater than", "not equal", etc. Symbol: "r".

(7) Unknown type: Symbol: "?". (initially, all tokens are typed as "?").

4.5.2 TOKEN SEQUENCES

Token sequences refer to acceptable token constructs that are read by the finite state automaton. The repertoire of the automaton may vary; however, all that is needed is the capability for verifying a sequence of tokens as to whether their syntax is correct. Thus, after the tokens are identified individually, the string is formed and then the pattern is verified.

The following patterns are a sample of these supported. Once a pattern is recognized, its family number is returned to the control procedure. With the family number, reorganization of the pattern is performed in order to further "formalize" (i.e., transform from natural to formal language) the query. Sample of patterns with examples and brief notation explanation is presented in Figure 17.

V (NB?)+ (VLB?)+	print names of students that live in "Dallas"
V (NB?)+ (NR+LB?)+	print names of faculty with salary greater than "24,000"
V (AN)+	print the good students
V (VLB?)	who is working in "Dallas"? ("who replaced with "retrieve name")

(a)	repetitions of construct "a"
a+	one or more occurences of construct "a"
a?	construct "a" is optional
a*	zero or more occurences of construct "a"

Figure 17 Sample Patterns and Queries

Patterns are less rigid in their requirements than other forms of NL representation such as ATN's [Winograd 83]. As a result, queries that do not conform exactly to syntactic standards can still be accepted, while acceptance of syntactically correct queries is not prohibited. The transformation mentioned earlier reformats the query so that it more closely resembles the SELECT-FROM-WHERE structure that is created by the formal query generation module. Restructuring typically involves grouping all conditional clauses together with their associated relational and conditional operators, and grouping of noun attributes.

4.5.3 SYNTACTIC VERIFIER IMPLEMENTATION

The syntactic verifier is implemented through the finite state automaton that recognizes the regular expressions that represent the RTN for each query. The UNIX operating system provides a lexical analyzer generator program, LEX, that accepts the specification for the patterns and possible actions desired and generates the finite state automaton that accepts such expressions, or rejects them. A meta-language is used in the specification of the patterns, with the associated actions embedded in "C". The result, after a pre-processing, is portable "C" code (or Fortran 77, if desired) that accepts or rejects regular expressions. [Lesk 76] describes LEX in more detail.

A sample regular expression recognizer is presented in Figure 18. The allowable constructs in the LEX meta-language are indicated below Figure 18. The actions have access to internal variables, such as the pattern length, current position of the match marker, etc., so that if the pattern fails, diagnostic messages can be issued. The example in Figure 18 recognizes simple patterns of variable names, integer and floating point types, and operators and returns appropriate token types to the scanner.

```
[A-Za-z][A-Za-z0-9_]*        { return (IS_VARIABLE); }
-?[0-9]+                     { return (IS_INTEGER) ; }
-?[0-9\.]+                   { return (IS_FLOATING); }
"+-*/%"                      { return (IS_OPERATOR); }
```

Figure 18 Sample LEX Scanner Specifications

Allowable constructs in LEX are as follows:

A-Z	matches single character uppercase
a-z	matches single character lowercase
0-9	matches single digit
[...]	groups sub-patterns
.	any character
*	zero or more times repetition
+	one or more times repetition
	indicates negation, also begin of line
$	indicates end of line
?	optional element

The output of the syntactic verification is either a pattern number indicating the family of patterns with which the input pattern was associated and recognized, or an error message specifying the location and nature of the error. In several cases, as mentioned earlier, slight token list transformation ("formalization") is performed. Then, the pattern number, the token list and the token identifier list (pattern) are passed on for semantic verification.

4.6 SEMANTIC VERIFICATION

Semantics refer to the meaning of words and word sequences. Semantic analysis refers to the analysis (and in the case of NL systems, verification) of NL input statements in order to verify their semantic correctness, based purely on semantic criteria [Wilks 82].

As [Dillon 83] reports, semantics are concerned with the meaning of entities. By meaning, he identifies the knowledge that an individual must possess in order to make judgements about ambiguity, anomalous construction, ellipsis and plethora, contradictions, redundant structures, equivalences and associations, and other concepts. However, while such knowledge is relatively adequate for human-to-human communications, it is not enough for NL question answering.

The process of semantic analysis and verification of input NL queries involves analysis of several non-linguistic concepts. The DBMS schema, for example, or a superset thereof, can be judged as a collection of abstractions on which application of knowledge can yield rejection or acceptance. Stonebroker suggests the addition of abstract data types and rule-based techniques for the INGRES database system [Stonebroker 76]. Other NL processing programs use semantic based representations (such as ATN's) for semantic verification.

KARL provides semantic analysis and verification capabilities related to both database system schema and linguistic considerations of the words in the knowledge base.

In the following sub-sections, both verification (and transformation, where applicable) techniques used in KARL will be presented.

4.6.1 <u>LINGUISTIC SEMANTIC ANALYSIS</u>

Handling linguistic semantic analysis involves a number of distinct operations on the input token list and token identifier list. At this time, the pattern family is known, as determined in the previous stage. The operations relate to verifying the query for inconsistencies that may arise from incorrect combination of terms, resulting in a query that is syntactically acceptable, but semantically incorrect. The tests that KARL is capable of performing on the input parameters include:

(1) Ambiguity is identified when multiple interpretations of a single term are found in the dynamic knowledge base. The main action is context analysis (i.e., lookup of the surrounding terms) and use of heuristics for determining the appropriate meaning. If context analysis fails, then the user is presented with the list of alternatives and selects to either proceed using one of these meanings, redefine the offending term or reject the query altogether.

C-2

(2) Ellipsis is identified when less terms are presented than_needed, i.e., when terms are missing from the query context. As with ambiguity, surrounding terms are used in order to identify the missing parts (or even attempt to "guess"), and then introduce the missing parts into the query structure.

(3) Redundancy is identified where duplicate information is given in a query context, i.e., in the query "give the names and names of students". Plethora is identified where more information than the required is supplied, i.e., "give the names of students and addresses of students". In both cases, the program will attempt to eliminate the useless terms.

(4) Relationships are verified following a set of rules encoded in the semantic verification module, and using dynamic knowledge as well. Under rules of the English language semantics, a number of groups of terms can be verified. Specifically, noun phrases, collections of nouns and/or noun modifiers (i.e., adjectives) are verified for compliance with the rules of concordance between adjective/noun structures. Also, in verb phrases, collections of subject/verb/object structures, the subject, verb, and object(s) have to agree.

Linguistic semantic verification is performed as a separate function within the system, with no interaction with the other semantic verification modules. Figure 19 illustrates the structure of the linguistic verification process:

```
              Token          |
              Flow           |
                             |
      +----------------------+----------------------+
      |                      |                      |
+-----v------+         +-----v-----+         +------v------+
| Ellipsis   |         | Plethora  |         | Ambiguity   |
+------------+         +-----------+         +-------------+
      |                      |                      |
      +----------------------|----------------------+
                             |
                             V
                  +----------------------+
                  | Verb Phrase Proc.    |
                  +----------+-----------+
                             |
                  +----------v-----------+
                  | Noun Phrase Proc.    |
                  +----------------------+
                             |
                             v
```

Figure 19 Linguistic Semantic Verification Flow Chart

4.6.2 DATABASE SEMANTIC VERIFICATION

A NL (or even formal) query to a DBMS can be analyzed and verified in terms of its semantics. The semantic correctness problem typically emerges when the query is syntactically correct, i.e., acceptable by the parser or DBMS front end, but the results are wrong, no response is produced, or an operating system level error occurs. In each case, analysis of the input query, usually using the schema or a superset thereof, can be used for semantic correctness verification.

KARL semantic verification of DBMS related entities (such as table organization, ranges, limitations, etc.) follows Stonebroker's suggestions for implementing semantics in the DBMS [Brodie 84]. Abstraction of the entities are stored in a schema superset, which in turn is represented in the frames of nouns in the dynamic knowledge base. The frames contain, for each abstract data type (essentially for each attribute), ranges, patterns, data types, relationships and operators allowable, and other constructs. KARL 1.02 database related verification follows the flowchart seen in Figure 20.

```
                Token          |
                Flow           |
                               |
           --------------------+------------------------
           |                   |              |            |
      +----v-----+      +-----v---+    +-----v---+    +---v---+
      | Literal  |      | Literal |    | Operator|    | Is-A  |
      | Ranges   |      | Patterns|    | Check   |    |Matches|
      +----------+      +---------+    +---------+    +-------+
           |                   |              |            |
           |___________________|______________|____________|
                               |
                               V
                  +---------------------+
                  | Operand Concordance |
                  +----------+----------+
                               |
                               V
```

Figure 20   DBMS Semantic Verification Flow Chart

Semantic verification in KARL involves passes over the token list and token identifier lists, with each pass verifying a distinct subset of the query as to its conformance to the correctness standards. The entire implementation can be divided into two parts, the linguistic and DB verification modules. The flow of processes throughout the two parts was presented earlier, in Figures 19 and 20. Figure 21 presents the integration of the two components into a discrete, autonomous unit.

```
         +--------------------------------------------+
         |                                            |
Token    |   +--------------+  Token  +--------------+ |
Flow     |   | DB Related   |  Flow   | Semantic     | |
-------->|   | Semantic     | ------->| Related      |---->
         |   | Verification |         | Verification | |
         |   +--------------+         +--------------+ |
         |                                            |
         +--------------------------------------------+
```

Figure 21 Integration of Semantic Verification Submodules

The routines that perform the semantic verification rely on semantic knowledge supplied by the knowledge base. Thus, given certain information (i.e., a verb), the knowledge base can return allowable subject/object combinations for linguistic verification. Similarly, given a noun (i.e., a database relation attribute), the knowledge base can supply all knowledge needed to verify its meaningful use.

In both sub-modules, knowledge is processed in a set of rules that accept the token and token identifier lists, pattern number and dynamic knowledge, and infer the compliance of the query to these rules. A sample collection of rules implemented through "C" language constructs is presented below using pseudo-English:

```
IF   TOKEN(N) IS ADJECTIVE
THEN TOKEN(N + 1) MUST BE NOUN AND   /* If not a noun  */
     NOUN AND ADJECTIVE MUST AGREE   /* a syntax error */
     AND HAVE ENTRY IN THE KB-ADJ.   /* is signalled   */
ELSE ERROR = NO-NOUN-ADJ-AGREEMENT.

IF   TOKEN(N) IS VERB                  /* K is the   */
THEN TOKEN(N-K), TOKEN(N+K) ARE NOUNS  /* lookahead  */
     AND MUST AGREE WITH THE DEFINI-   /* and/or     */
     TION OF THE VERB IN THE KB-VERB.  /* backtrack  */
ELSE ERROR = NO-VERB-NOUN-AGREEMENT.   /* pointer    */

IF   TOKEN(N) IS LITERAL
THEN TOKEN(N-K) IS THE NOUN ENTITY
     SO VERIFY THAT LITERAL RANGE  /* i.e., GPA = 6.0 */
     IS ACCEPTABLE
ELSE ERROR = LIT-OUT-OF-RANGE.
```

Figure 22 Sample Semantic Verification Rules

Implementation of the rule structure itself is made using the "C" programming language, and, in effect, constitutes a part of the static knowledge of the system. Such knowledge (i.e., knowledge that a literal must be within a certain range of high and low values, or that a noun phrase's components must agree) is typically independent of applications and can be reused as applications vary.

4.7 FORMAL QUERY GENERATION AND EVALUATION

After having processed the NL query and verified its syntactic and semantic correctness according to predefined criteria, a formal query has to be generated and executed by the host DBMS. Although the two operations are rather distinct, their combination is necessary so that host DBMS related dependencies are minimized. This was one of the generic design goals, and the solution is well suited in fulfilling the objective.

In generating and evaluating the formal query from the NL or transformed NL query, two approaches can be considered. One is to use the low-level DBMS facilities, where applicable, and make the NL processor responsible for interfacing the formal query generator with the low-level DBMS routines that perform the actual retrieval. The other option is to allow the NL processor to evaluate the generated formal query as if it were a user in an interactive session, by typing in the formal query to the high-level DBMS interactive monitor. This approach would allow better portability as many formal query languages for relational DBMS's tend to be similar and simplify the generation part to a great extent. On the other hand, interface with the low-level DBMS services would imply layering and/or other CPU intensive processes in order to determine the proper sequences of subroutine calls that are needed for evaluating the query.

Both methods were considered in the design of KARL. As the generic as well as specific design objectives call for simplicity and portability, the second approach of interfacing at a high level with the DBMS was adopted. The process of transforming the NL processed input token list into a formal query for the INGRES relational database system is outlined below:

(1) Determine the domains and ranges of the NL query, and the abbreviated names that are to be used in the RANGE statement.

(2) Determine the type of operation (retrieval, existence check, count, etc.) that is being requested and verify that the request is supported (in version 1.02, only retrieval is supported).

(3) Select the attributes that are to be retrieved, or use the defaults from the database schema frame representations. If none is specified, default to ALL and prepare full tuple retrievals.

(4) If no conditionals are specified, submit the query after reorganization to conform to the formal query structure of the host DBMS system.

(5) If there are conditionals and possibly boolean conjunctions, determine the conditional parts of the transformed NL query. Use the patterns of <noun_phrase> <relational_op> <literal> structures in order to determine the exact conditions that are to be met. Using a "blank" formal query structure, perform the "fill in the blanks" operation for each conditional statement/pair. Link the conditionals with the appropriate boolean connectors (and, or, not). Once the formal query has been filled in, proceed with next step.

(6) Check the entire formal query for syntactic correctness using knowledge of the host formal language syntax. If correct, prepare for evaluation, else flag the query as incorrect due to internal (not user-related) error.

(7) Perform the necessary calls to the host DBMS to open the DB, submit the query, and then, after the results have been presented to the user, close the DB and proceed. This concludes the query cycle.

The blank formal query structure mentioned earlier has the form of Figure 23. The attribute list, domain list and conditional lists are present, with the conditional statements being optional. Transformation of such a query structure to the QUEL query structure is simple, since all three of the necessary information subsets (i.e., domains, attributes, and conditionals) have already been determined.

```
    "Blank" Format:

        SELECT    <attribute_list>
        FROM      <domain>
        WHERE     <condition_list>

    QUEL Format:

        RANGE OF <abbrev_name> IS <domain>
        RETRIEVE <dot_attr_list>
        WHERE    <dot_conditional_list>

    <attribute_list> ::= <attribute> | <attribute_list>
    <attribute>      ::= any database relation column name
    <domain>         ::= any database tuple name
    <condition_list> ::= <attribute> <rel_op> <literal> |
                         <attribute> <rel_op> <literal> <bool> |
                         <condition_list>
    <abbrev_name>    ::= shortcut name used in retrievals
    <dot_attr_list>  ::= <abbrev_name>.<attribute> |
                         <dot_attr_list>
    <dot_cond_list>  ::= conditional list using dot pairs
```

Figure 23 "Blank" and QUEL Formal Query formats

At this point, it is reiterated that the transformation of the user input NL sentence into a fully formal query does not occur at a single stage, but rather at different points during syntactic and semantic analysis. For example, after verifying the semantic correctness of a noun phrase consisting of adjective/noun pairs, the semantic verification module replaces the noun phrase with the more formal <attribute> <relational_op> <literal> structure. Similar transformations occur if elliptic queries are being processed, when, after determining the missing terms, the terms are inserted into the query and the token and token identifier lists are modified to reflect the new transformations.

The formal query generated for the host DBMS is evaluated and processed by the host DBMS itself, thus simplifying even more the task of NL processing. After the formal query is submitted, the host DBMS will respond in the same manner it would as if the formal query were typed on the terminal monitor. Then, the results are displayed on the user terminal screen and the cycle is set up again.

4.8 MODULE INTERCONNECTION

Module interconnection refers to both the connections of the various (loosely coupled) modules with each other, in order to construct the entire system, and also, the connections of the entire software system with the underlying operating system, DBMS and run-time support environment. Both interconnection schemas will be presented.

4.8.1 INTERNAL CONNECTIONS

Aho [Aho 79] suggests the schema of Figure 24 for a compiler. As illustrated, a pipeline-like structure accepts the user program in one end and returns object code (and possibly error messages) from the other end.

```
                         Input Program
                               |
                               |
                    +---------V--------+
                    | Lexical Analysis |
                    +------------------+
                               |
                               |
                    +---------v--------+
                    | Syntax Analysis  |
                    +------------------+
+---------+                    |                  +--------+
| Tables  |                    |                  | Error  |
| Manage- +         +---------v--------+          + Handl- |
| ment    |         | Intermediate Code|          | ing    |
+---------+         | Generation (some)|          +--------+
                    +------------------+
                               |
                               |
                    +---------v--------+
                    | Code Optimization|
                    +------------------+
                               |
                               |
                    +---------v--------+
                    | Code Generation  |
                    +------------------+
                               |
                               V
                        Executable Code
```

Figure 24 Typical Compiler Organization

As KARL shares a number of features and concepts from compiler design, a similar structure that couples together the modules that were presented earlier can be visualized. The main parts, as seen in Figure 25, are the lexical/grammar analyzer, syntax verifier, semantic verifier and formal query generation and evaluation modules.

```
                         Input NL Query
                               |
                               |
                     +---------v--------+
                     | Lexical and      |
                     | Grammar Analysis |
                     +------------------+
                               |
                               |
                     +---------v--------+
                     | Syntax Analysis  |
                     | and Verification |
                     +------------------+
+-----------+                  |                  +--------+
| Knowledge |                  |                  | Error  |
| Base Mgmt +        +---------v--------+         + Handl- |
| System    |        | Semantic Analysis|         | ing    |
+-----------+        | and Verification |         +--------+
                     +------------------+
                               |
                               |
                     +---------v--------+
                     | Query Generation |
                     +------------------+
                               |
                               |
                     +---------v--------+
                     | Query Evaluation |
                     +------------------+
```

Figure 25 KARL Structure Organization

There are three distinct flows of data (machine-readable code) in the KARL system. The query tokens flow, the knowledge flow, and the errors/warnings flow. The three flows can be thought of as complementary, as no functions overlap and each module has a determined task allocated.

(1) Flow of tokens involves the "movement" of tokens from the initial user interface prompt stage to the final

stage_of processing by the host DBMS formal query parser. The NL query that is input passes through a number of transformations, each formalizing the query and updating the token identifier list. The token list is initiated in the lexical and grammar module. Then, after identification of the tokens, the token identifier list "travels" with the token list. Syntactic verification generates one additional item of information, the pattern family number, which is also forwarded for semantic analysis and formal query generation purposes.

(2) Flow of knowledge is bidirectional from the dynamic KB to and from the individual modules that use it. Such knowledge can be dictionary knowledge, i.e., word classes, and semantic knowledge that relates to the attributes and overall schema of the database. Static knowledge is considered local to the individual modules and is of no concern at this phase.

(3) Error and warning flow is unidirectional from any module in which an abnormal condition may arise. Errors are identified as functions of the following parameters: original NL token list, processed token list, token identifiers list, error number, and error message. Typically, the program that issues the error message does not take action, but turns control over to the

error_handler and reporter module. This module can also contain diagnostics for the user. Finally, warnings are handled similarly to errors, but do not abort the query processing cycle.

4.8.2 EXTERNAL CONNECTIONS

External connections are the interaction paths between KARL, INGRES and UNIX. Figure 26 presents the relationship of the three products. All data paths between components are bi-directional.

```
+-----------------------------------------------------------+
| UNIX |                                                    |
|------'                                                    |
|                                              +--------+   |
|                                              | System |   |
|        +-----------------------------+      | Calls  |   |
|        |                             |      +---+----+   |
|        | The KARL System             |          |        |
|        |                             +----------+        |
|        +------+-------------------++----+                 |
|               |                   ||                      |
|      low      |                   || high                 |
|     level     |                   || level                |
|               |                   ||                      |
|        +------+-------------------++----+                 |
|        |  INGRES Relational DBMS        |                 |
|        |                                |                 |
|        +--+-----------+----------+---+                    |
|           |           |          |                        |
|           |           |          +----------+             |
|           |           |                     |             |
|    ::::::::::  ::::::::::          XXXXXXXXXXXXX          |
|    :: data ::  :: data ::          X Knowledge X          |
|    :: base ::  :: base ::          X   Base    X          |
|    ::::::::::  ::::::::::          XXXXXXXXXXXXX          |
+-----------------------------------------------------------+
```

Figure 26 Inter-System Organization

Communication between INGRES and KARL occurs through two primary methods. The first is through embedded INGRES statements within KARL code, in the Embedded QUEry Language (EQUEL) that INGRES supports. In such, INGRES QUEL statements are placed (embedded) within "C" code, and then a preprocessor translates them into INGRES low-level calls. The second approach involves direct calls of the low-level INGRES capabilities, usually in order to overcome the inherent difficulties present within EQUEL. It is noted, however, that most of the interfacing is performed through EQUEL, and only the critical parts are implemented directly through INGRES calls. Transportability is not affected significantly since the embedded query capabilities of many relational DBMS's, like the SQL/System R embedded query language [Date 81], are similar to the one used by INGRES in KARL.

The second level of interaction is between KARL and UNIX. UNIX supplies information to KARL through system calls. Such services are date, user id, access information, etc. For portability reasons, only the functions that are available in a variety of operating systems (such as time and access information) are used. [Kerningham 79] contains additional information on the interaction of application programs and UNIX.

4.9 ANNOTATED EXAMPLES OF QUERY PROCESSING

This section will present several annotated examples of queries that were processed and/or rejected by KARL. For each query, the pattern and the different stages of processing will be explained. In total, six queries will be presented and discussed. Three failed and three were accepted by the system.

QUERY 1:

```
   please show the students enrolled in "CMPS351" or "CMPS360"

   LEXICAL ANALYSIS: show student enroll "CMPS351" or "CMPS360"

          (ellipsis): show student enroll "CMPS351" or
                                  enroll "CMPS360"

   PATTERN  MATCHED: Verb (Noun Bool?) (Verb Literal Bool?)*

   SYNTACTIC ANALYSIS: OK. Pattern Accepted, Pattern_No = 8.

   SEMANTIC ANALYSIS: enroll (student, course)      OK.
                      course PATTERN = "XXXX9999"   OK
                      course Number  = 360 < 699    OK
                      course Number  = 351 < 699    OK

   BLANK QUERY:       SELECT all  /* default */
                      FROM   student
                      WHERE  (course = "CMPS351" |
                              course = "CMPS360" )

   QUERY PROCESSED CORRECTLY
```

Example 1 Query With Simple Ellipsis

Example 1 was processed with qualifying attribute ellipsis for the literal "CMPS360". As in programming languages, the previous attribute is used by default.

QUERY 2:

```
    who is "000-4076-65"

    LEXICAL ANALYSIS: retrieve name "000-4076-65"
    (severe ellipsis): retrieve name "000-4076-65"

    PATTERN  MATCHED: Verb ( Noun Rel_op? Literal Bool? )+

    SYNTACTIC ANALYSIS: OK. Pattern Accepted, Pattern_No = 4.

    SEMANTIC ANALYSIS: Pattern "999-99-9999" matches ssn
             REFORMS: show student ssn "000-4076-65"
                      ssn  PATTERN = "999-9999-99"   OK

    BLANK QUERY:      SELECT name
                      FROM   student
                      WHERE  (ssn = "000-4076-65")

    QUERY PROCESSED CORRECTLY
```

Example 2   Query With Severe Ellipsis

QUERY 3:

```
    print names and addresses of all the rich faculty

    LEXICAL ANALYSIS: print name address rich faculty

    PATTERN  MATCHED: Verb (Noun Bool?)+ ( Adjective Noun )+

    SYNTACTIC ANALYSIS: OK. Pattern Accepted, Pattern_No = 12.

    SEMANTIC ANALYSIS: name belongs to faculty
                       address belongs to faculty
                       rich := salary > 40,000
             REFORMS:  print name address faculty salary > 40000
                       salary range OK
    BLANK QUERY

             SELECT name, address
             FROM   faculty
             WHERE  salary > 40000

    QUERY ACCEPTED
```

Example 3   Query With Adjective and Noun

Example 2 presented severe ellipsis which can be handled when the appropriate number in the pattern family is determined. Then the literal patterns frame is scanned and the "student" frame has that pattern. Example 3 uses adjectives as noun modifiers, and the semantic verifier uses the adjective and verifies its use with the noun. Both queries are accepted.

```
QUERY 4:

    show students who live and work in "Lafayette"

    LEXICAL ANALYSIS: show student live and work "Lafayette

    PATTERN MATCHED: NONE (although sentence is correct)

    SYNTACTIC ANALYSIS: Failed. Program could not parse
                        input sentence (No double verb
                        pattern supported)

    QUERY REJECTED
```

Example 4 Query With Non-supported Pattern (Two Verbs)

```
QUERY 5:

    show the rich students

    LEXICAL ANALYSIS: show rich student

    PATTERN MATCHED: Verb ( Noun Relop Literal Bool?)+
    (severe_ellipsis, pattern matches after replacing "rich")

    SYNTACTIC ANALYSIS: OK. Pattern valid. Pattern No: 4

    SEMANTIC ANALYSIS: rich student: error.
                       Attribute "salary" not associated with
                       relation "student"

    QUERY REJECTED
```

Example 5 Query With Incorrect Semantics (Adjective)

QUERY 6:

```
    show the students enrolled in "CMPS999"

    LEXICAL ANALYSIS: show student enroll "CMPS999"

    PATTERN  MATCHED: Verb ( Verb Literal Bool? )+

    SYNTACTIC ANALYSIS: OK. Pattern valid. Pattern No: 11

    SEMANTIC ANALYSIS: enroll (student, class) OK
                       class pattern OK
                       class number out of range
                       class number > 699
    QUERY REJECTED
```

Example 5 Query With Incorrect Semantics (Range)

Some of the examples that failed were erroneous because of range, syntax, or adjective/noun concordance (Examples 5 and 6). There are other reasons that queries fail, in particular queries that are out of the program's capabilities (Example 4). Such queries and future work are discussed in Chapter 5.

4.10 CHAPTER CONCLUSIONS

In this chapter, the low level design and implementation of the KARL software system were presented. The system design was decomposed into its functional modules, and each module was presented and discussed as an independent entity. The interconnections between modules were also presented and discussed.

Although technical details in a design of such complexity are typically overwhelming, the modular design of KARL assisted in presenting the design itself as well as the underlying concepts in a structured way. The methodology that was followed in the design was also stressed.

Implementing a software system as diverse and as complex as KARL was an experience in itself. Being able to materialize the theoretical concepts underlying KARL (RTN's, database theory, compiler theory, formal languages, linguistics) into a single functioning software system indicates that the integration of the concepts was far more difficult than either the selection of design techniques or implementation techniques. In such an environment, the need for controlling the interaction between independent components was critical, and the presence of a single methodology for integration was appreciated. Then, by integrating the various components, full functionality was achieved.

CHAPTER 5

DESIGN EVALUATION AND FUTURE ISSUES

5.1 DESIGN EVALUATION OBJECTIVES

Completion of the design and implementation phases of a software product is not considered the end of the software life cycle [Turner 84]. Product evaluation, based upon the product's own design objectives, user opinions and accepted standards are all needed in order to determine the success and/or failure of the product. Evaluation based on these criteria is presented in this chapter.

The changing field of natural language query processing systems, combined with current progress in interdisciplinary areas such as human-machine interaction studies, linguistics, and cognitive psychology, create the need for a design that not only performs according to set standards, but is able to expand in order to accommodate new techniques, modifications or improvements. As one of the principles of KARL is its expandability, a framework for future expansions is presented in this chapter. The framework contains, as examples, several proposals for major upgrades that originated during the design and implementation phases of the prototype version 1.02.

5.2 EVALUATION OF GENERIC OBJECTIVES FULFILLMENT

The generic objectives were general guidelines to be followed in the design of KARL. These were general objectives that can apply to any software system, and thus they were adopted for a NLQS. As design objectives, these characteristics indicate the main areas of attention of the designer. The objectives, in order of importance, were as follows:

(1) Adaptability to new applications

(2) Portability between systems/host tools

(3) Reduced complexity

(4) Efficiency.

Using these objectives as guidelines for system design, the high level design of KARL was undertaken. Comparing the generic design objectives with the results, from both the design and implementation phases, it is evident that the generic objectives have been fulfilled:

(1) Adaptability to new applications has been achieved by providing a fully modifiable dynamic knowledge base that contains the application dependent knowledge. Therefore, new applications only need redefinition of the dynamic knowledge base contents. Although building a new

knowledge base is by no means trivial, even in very limited expertise domains [Wiederhold 84], it certainly is less resource- and time-consuming than having to modify the system implementation or develop new applications.

(2) Portability between systems/host tools has been achieved through the use of a widely available host operating system (UNIX), programming language (C) and host database system (INGRES). In addition, the source code is portable (i.e., contains no major operating system calls except the "system" call that passes a command line to the operating system from execution within a program, which is a facility available on most modern operating systems), and the structure of the INGRES query format can be retargeted to other relational DBMS's without significant effort. Finally, the lexical analyzer generator creates portable C code.

(3) Reduced complexity is also achieved. This is evident in the source code that is divided into logically distinct modules performing independent tasks. Reduced complexity therefore is the result of a "black box" design methodology, rather than the result of implementing a particular NL processing strategy.

(4) Efficiency was also achieved through the use of a highly

optimized compiled language instead of a more traditional AI-oriented interpreted language. The program's simple structure eliminates many calls to routines performing multiple functions and contains no dynamically allocated memory, thus optimizing the implementation even more. Should the efficiency and performance become critical, migration to a memory-resident knowledge schema can eliminate the overhead caused by the knowledge retrieval process.

5.3 EVALUATION OF SPECIFIC OBJECTIVES FULFILLMENT

The following natural-language specific objectives were identified when the framework for the design and implementation of KARL was presented:

(1) System knowledge capabilities and its associated domains, processing, and acquisition.

(2) Grammatical constructs handling capabilities that allow recognition of different forms of the same word; also, capability of handling synonyms.

(3) Syntactic construct handling capabilities that allow recognition of different syntactic forms of questions.

(4) Semantic construct handling capabilities that allow verification of different semantic forms of questions.

(5) Learning capabilities that allow a system to "learn" new words and constructs.

(6) Handling of elliptic queries, thus necessitating heuristics in order to understand and process such queries; also capabilities for generalized error detection and appropriate reporting.

Using the specific design objectives as criteria for the evaluation of the design, the following is a list of the status of these objectives:

(1) System Knowledge capabilities and associated domains, processing, and acquisition are provided through the dynamic knowledge definition procedures, the knowledge utilization procedures and the embedded rules contained in the static parts of the knowledge base.

(2) Grammatical constructs handling capabilities that allow recognition of different forms of the same word and capability of handling synonyms are provided through a collection of grammatical transformation rules and the dictionary which allow multiple forms represented as one entry and multiple synonyms mapped to the same entry also.

(3) Syntactic construct handling capabilities that allow recognition of different syntactic forms of questions

are provided. Such capabilities allow the verification of sentences based on syntactic criteria, using a simple network-based algorithm that allows English, pidgin-English and even semi-formal queries to be recognized.

(4) Semantic construct handling capabilities that allow verification of different semantic forms of questions have been provided. Semantic verification at both the database level and the linguistic level are provided, along with descriptive diagnostics.

(5) Learning capabilities that allow a system to "learn" new words and constructs are provided also. When terms that are unknown to the system (i.e., not in the dictionary) are encountered, the system has the capability of querying the user and then retaining the answer for future use, thus providing a form of learning.

(6) Handling of elliptic queries is provided through context analysis. The surrounding context is used in order to determine the missing terms, along with defaults set in the dictionary and heuristics where appropriate. Full diagnostics are also provided for all stages.

The specific objectives are met primarily through the following of a decomposition methodology that allows the designer to concentrate on one specific part of the system (or objective). Such approaches are typical of large scale design and implementation projects such as compilers, and the "black box" methodology and presence of a single intermediate query representation form ensure compatibility between the modules.

5.4 FUNCTIONAL EVALUATION OF SYSTEM PERFORMANCE

A system is evaluated in order to determine not only how well it conforms with the original design objectives and specifications, but also in order to determine its overall functionality and capabilities in handling the task(s) for which it was designed.

In evaluating software products in general, in order to determine their functional capabilities, the problem of adequate testing is often addressed [Wernier 79]. There are suggestions for both basic and advanced level testing. In more traditional software systems, where the set of possible inputs is not infinite, testing can often determine the success or failure of the system by using as many cases as possible and observing the results. Even in the cases of software systems as complex as the Ada compiler, there are test case collections or suites, that have to be executed in order to verify the correctness of the system [Barnes 84].

In the field of natural language query systems, however, validation of the correctness of the program has been overwhelming difficult. There are no input restrictions, and usually a wide set of rules that are imposed (i.e., the database schema) exist. A NL processing system, therefore, can not be verified solely in terms of its input/output alone. Although other software systems can be said to function/malfunction solely on the basis of their input and producing output, NL systems can not be verified by solely typing queries to the program and counting the number of successes and failures.

It is then concluded that a NL system can not be judged in terms of sample inputs/outputs alone. Tests of earlier versions of KARL (KARL 1.00) indicated a capability of handling queries in the 60 to 65 percent margin, when adjusted for spelling and typing errors. However, judging the overall functionality of the system not in terms of the percentage of queries that it handled, but rather by comparing it to accepted criteria for NL processing systems is more appropriate. This serves as an evaluation of the design concepts, methodology and techniques rather than an evaluation made on "looks alone", i.e., how the system responds to the end user.

The set of NL processing capabilities that was defined by Hendrix [Hendrix 81] is used to evaluate the general performance of the KARL system. The capabilities, presented in Figure 27, characterize a production-level NL database query system in terms of its capabilities and design provisions/characteristics that are incorporated into it. The criteria are not to be taken as the only means of determining success or failure, but can be used as guidelines towards that decision. The set of the capabilities, along with KARL's performance "ratings", can be seen in Figure 27.

	CRITERION	KARL
(1)	Be able to access multiple databases (i.e., retargetable within applications)	YES
(2)	Answer questions asked directly (i.e., Who ...)	YES
(3)	Handle multiple files and relationships	YES
(4)	Handle simple pronoun references	NO (i)
(5)	Be able to handle ellipsis	YES
(6)	Provide report generating facilities for the retrieved data (i.e., formats, graphs, etc)	NO
(7)	Be able to extend the linguistic knowledge of the system during program execution (Learn)	YES
(8)	Handle null (no retrieval) cases, indicating the condition(s) that failed	NO (ii)
(9)	Restate in English the user's query, to assist in understanding the system's view of the query	YES (iii)
(10)	Handle spelling and typing errors caused by users	NO
(11)	Provide special functions for improvement of the database capabilities	NO (ii)
(12)	Provide semantic constraints in the dialogue between the human and the machine, and handle errors such as plethora and ambiguity	YES

(i) Item has been considered as future extension (next Section)
(ii) Item not in the original design considerations
(iii) The program restates the query in a semi-formal way.

Figure 27 Hendrix's Capabilities and KARL performance

In addition to general natural language capabilities discussed earlier, a NLQS can also be evaluated in terms of the subset of the natural language that it is capable of handling properly. This language subset can be considered as the union of common concepts that can be used by the user and the linguistic facilities that describe how these concepts can be expressed [Tennant, 80].

Both aspects of the natural language are important; furthermore, the integration of concepts and facilities must be made in such a way as to ensure maximum linguistic performance. The concepts that the current version of KARL is capable of handling are presented below in an outline form proposed by [Tennant, 80].

Common Natural Language Concepts

- Closed class words
  - definite references
  - gender
  - number
    - counted objects
    - singular/plural
  - modality
  - location
    - position
    - general area
  - time
    - past/present/future
    - different representations
    - time span (interval)
  - possesion/ownership
- Domain-Specific Concepts
  - database elements
    - fields, attributes, values
    - relationships between fields
    - restrictions/limitations
  - application knowledge
    - domain-specific knowledge
    - knowledge extension

Logical Relationships
- negation
- disjunction
- conjunction

Quantative Relationships
- numerical quantifiers
- character string quantifiers
- comparison

Extension of Concepts
- equivalence terms
  - synonyms/acronyms
- new classes
  - named classes
  - named objects
  - named properties

Linguistic Facilities

Concept Reference Capabilities
- by name (string constants)
- by class
  - modifiers
  - determiners
  - quantifiers
  - identifiers
- by adjective classes
  - adjective phrases
  - adjective/noun phrases
- by action indicators
  - verbs
  - verb phrases
- by other means
  - synonyms/antonyms
  - acronyms
  - property lists
  - numeric values

Sentence Structure
- active voice
- limited passive voice
- simple sentences
- multiple sentences
- declarative/imperative/interrogative
- noun phrases
  - subject-verb-object type
  - limited indirect type
  - multiple noun phrase type
- finite verb phrases
- non-finite verb phrases
- verbless clauses
- adjective phrases
  - prenominal phrases (adjective-noun)
  - postnomial phrases (noun-adjective)

```
        multiple adjective sequence
  Elliptic Phrases
      ellipsis and substitution
      nominalized adjectives
           assume noun from adjective
      nominalized verb phrases
           assume noun from verb
      ommited conjunctions/sub-sentence connectors
```

5.5 CURRENT STATUS AND FUTURE WORK

The initial implementation of KARL was made in order to determine the validity of the design concepts, namely, the highly independent processing modules, the relational implementation of the knowledge base, and the system's capability to retarget. As a result, a number of features present in production-level systems have not been implemented. This section will present a collection of such features, along with a framework for future design and implementation.

The collection of features that can be implemented in a NL system can easily become extremely large, as there are always new rules, features, and improved capabilities that can be added, or even old ones that can be replaced/improved. The key aspect in this type of system upgrade is the expandability of the system. KARL, based on a number of independent modules and a simple representation of knowledge and intermediate query form, can be expanded by replacing modules, or adding new modules between modules. Since the functionality of each module is well defined, a future replacement can integrate the updates in the existing frame with little effort. Should additional modules be needed, such as a spelling checker, they can be added between modules.

In its current status, KARL 1.02 is targeted towards a simple student/faculty/course database. The configuration of the database schema contains four relations. None of the knowledge required to process queries on the database is hard-coded, and all is contained in the dynamic part of the knowledge base. The knowledge base itself occupies approximately 18 Kbytes of storage (dynamic part only). A copy of both the database schema and contents, and the knowledge base schema and contents for the sample application used throughout the thesis can be found in Appendix A.

KARL does not support nested queries, therefore it can process only queries related to one relationship at a time. Also, it does not handle spelling or typing errors due to time limitation considerations. KARL's capabilities for processing null queries handle only cases where a null response is the result of a semantic error, not cases where the conditional is correct but there is no such value (or values) in the database. Finally, the prototype version 1.02 does not handle pronoun reference. A framework for designing and integrating these capabilities within KARL is presented below:

(1) Nested queries can be implemented by recursively selecting the conditions for the sub-queries, or alternately, defining the maximum number of levels of sub-querying and iteratively constructing the query. In order to group the elements of the different sub-queries into one structure, the query generation module will have to be expanded to accommodate multiple conditionals in the WHERE part of the query. The same syntactic and semantic constraints will apply.

(2) Pronoun reference can be handled by maintaining a query history and applying the criteria for the most recent query that agrees semantically with the pronoun-referencing query in question. For example, a query to display a certain student's record followed by a question of the form "When did he take CMPS550?" could be answered easily. Should other queries interleave, heuristics that match the rest of the attributes of the pronoun-referencing query to the ones previously in the history would be used. Pronoun reference handling capabilities can be inserted after the lexical stage so that the query is fully resolved for syntactic/semantic analysis issues. User querying can be considered as a "last resort" solution. The other system components would need no changes.

(3) Spelling correction can be handled in several ways; one way would be to assign unique similarity codes to words and then fetch words of similar similarity codes as alternative(s). Heuristics can be used to correct several types of errors, for example, extrapolating characters, or forgetting to type a space between words, or removing one character from the word. However, correction of spelling errors requires relatively large resource utilization, and tradeoffs have to be made for system capability versus processing time. The spelling check/correct module can be attached to the lexical analysis stage with no modification to the remaining components of the program.

(4) Null query handling capabilities involve decomposing the query and re-submitting the fragments for execution, noting the number of hits. This feature can be added after the query evaluation stage, and be activated when a null result occurs. Decomposing the query would involve boolean processing capabilities and techniques which do already exist in the field of compiler construction. Heuristics can be used so that if a part of the query that fails affects others (i.e., through an AND construct), the search for the null-causing clause terminates. It should be noted that with the range and pattern semantic verification capabilities, and a

well-planned database, user errors that result in null queries are reduced. No changes to the program structure would be required.

(5) Query optimization is another area that future research can address. Using application dependent knowledge, the query processor can eliminate conflicting clauses or simplify queries to a large extent. Considerable research has been undertaken on the subject [Wiederhold 84]; optimizing should be targeted towards the formal language query, since the "unstable" NL query can not be formalized enough before optimization. In addition, code optimization techniques can be used. Such a module would be an extension of the formal query generation module, with no changes required to other program modules.

The field of NL processing by computer offers highly challenging problems. Orienting the product towards production use brings into consideration computational efficiency as well as NL handling capability. Finally, user surveys can be used in order to determine needed system qualities and, through software maintenance, introduce these into the system.

CHAPTER 6

CONCLUSIONS

In this thesis, the design and implementation of a knowledge assisted restricted natural language database query system, the KARL system, have been presented. The general methodology, as well as the specific techniques that have been followed throughout the research have been explained. Future areas for research have also been identified, using the KARL system as a foundation and research vehicle.

The significance of this thesis is twofold: First, a design methodology for the construction of a NL query system for DBMS systems has been presented. With the increasing applications of computerized information systems in everyday life, there is a definite need for such systems. In addition, the methodology and specific techniques described in the thesis can be adapted for use by other applications software front-ends or by integrating DBMS's and other applications software under a common NL interface.

The second significant fact is that a NL database front end has been designed and implemented using primarily common techniques found in Computer Science. General methodologies that have been proven effective by years of experience are used in the thesis' high-level and low-level design and implementation. The result is a software product that is adaptable to new applications, has a high degree of transportability between environments, and is relatively simple (by means of a highly decomposed structure) to understand.

Although it seems unlikely that, within the near future at least, computers will be able to communicate fully in natural language in a way similar to HAL-9000, decomposing the problem into smaller, more solvable areas such as speech recognition, abstracting, indexing, and natural language query processing, can create an environment where, by integrating all the sub-elements, a full scale natural language processing computer can be realized.

The methodology that has been followed has been used commonly in production software development environments. However, NL development efforts have had a tendency of being highly individualistic, with large scale, difficult to maintain programs being the rule [Wasserman 85; Eisenberg 84]. Functional decomposition allows the designer to concentrate on one part of the problem, while the independent construction of the modules ensures that side effects are minimized and/or controlled. The use of a commonly available operating system (UNIX), a common database system (INGRES) and a common programming language (C), ensures that the techniques-can be applied to similar, non-NL or non-AI specific environments.

The methodology followed in this thesis attempts to solve the problems by using a "Computer Science" rather than a "Human Language" approach. So far, attempts to emulate or simulate the human perception of language have met with mixed results, and overwhelming efforts [Coomps 81; Lehnert 78]. This thesis approaches the problem of translating NL input queries to formal queries by decomposing the problem into its distinct parts and applying existing solutions (where applicable, i.e., lexical analysis, query generation) or developing such solutions using integrated environments (UNIX) and tools (LEX).

It has been said that there will not be a human-made machine that can simulate a bird's flying. The fact that humans have not achieved this feat does not limit them from flying at speeds many times the speed of birds. Under the same methodology of being inventive rather than attempting to simulate nature, computers may never achieve simulation of the human process of thought, but, as with airplanes, new techniques can be invented that achieve the end result and even outperform nature to a great extent. In the case of NL processing, the two prime candidate approaches that have been followed so far are emulation of the human's perception of language using linguistic and cognitive psychology models

[Lehnert 78], and the computer-based approach of functional equivalence rather than simulation [Embley 85]. This thesis followed the second approach.

Neither approach has been completed thus far. This thesis has proposed solutions to some of the basic problems of NL processing by computers. As there are many more areas in which solutions can be addressed, this thesis has also presented and identified future research issues. Utilizing presented methodology, existing systems, and integration techniques available from today's software development facilities, future research can proceed (hopefully a bit faster) into the widely desired end product, the true "human computer".

REFERENCES

[Aho 78] Aho, Alfred V. and Ullman, Jeffrey D, Principles of Compiler Design, Addison-Wesley Publishing Co., Reading, MA, 1978.

[Barnes 84] Barnes, Jean-Paul G., Programming in Ada, Addison-Wesley Publishing Co., London, England, 1984.

[Blanning 84] Blanning, Robert W., "Conversing with Management Information Systems in Natural Language", Communications of the ACM, Vol. 27 No. 3, pp. 201-206, 1984.

[Brachman 83] Brachman, Ronald, "What IS-A Is and Isn't: An Analysis of Taxonomic Links", IEEE Software Magazine, Vol. 18 No 10, pp. 30-36, 1983.

[Brodie 84] Brodie, Michael L., John Mylopoulos, and Joachim Schmidt (Editors), On Conceptual Modelling, Springer-Verlag, New York, NY, 1984, 510p.

[Brown 76] Brown, P. J. (Editor), Software Portability, Cambridge University Press, London, England, 1976, 328p.

[Bolc 83] Bolc, Leonard (Editor), The Design of Interpreters, Compilers and Editors for Augmented Transition Networks, Springler-Verlag, Berlin, Germany, 1983, 214p.

[Cater 83] Cater, A., Semantic Problems in Parsing, J. Wiley and Sons, New York City, NY, 1983.

[Charniak 76] Charniak, Eugene, Syntax in Linguistics, in Computational Semantics, Eugene Charniak and Yorick Wilks, editors, North-Holland Publishing Co., New York City, NY, 1976.

[Clarke 73] Clarke, Arthur, 2001: A Space Oddissey, p. 385, Penguin Books, New York City, NY, 1973.

[Coomps 81] Coomps, M. J., and J. L. Alty (Editors), Computing Skills and the User Interface, Academic Press, New York, NY, 1981.

[Dahl 83] Dahl, Veronica, "Logic Programming as a Representation of Knowledge", IEEE Software Magazine Vol. 18, No. 10, pp. 106-111, 1983.

[Date 81] Date, J. C., An Introduction to Database Systems, Volume I and II, Addison-Wesley Publishing Co., Reading, MA, 1981.

[Date 83] Date, J. C., Database: A Primer, Addison-Wesley Publishing Co., Reading, MA, 1983.

[Dillon 83] Dillon, George L. Introduction to Contemporary Linguistic Semantics, Prentice-Hall Inc., Englewood Cliffs, NJ, 1983.

[Eipstein 79] Eipstein, Robert, A Tutorial on INGRES, pp. 1-28, ECL, University of California-Berkeley, Berkeley, CA, 1977.

[Eisenberg 84] Eisenberg, Janet and Hill, Jeffrey, "Using Natural Language Systems on Personal Computers", Byte Magazine, Vol. 8 No. 1, pp. 226-238, 1984.

[Embley 85] Embley, David W. and Kimbrell, Roy E., "A Scheme-Driven Natural Language Query Translator", Proceedings of the ACM Computer Science Conference '85, 19p, 1985.

[Freeman 81] Freeman, Peter and Anthony I. Wasserman (Editors), Tutorial on Software Design Techniques, IEEE Publications, Long Beach, CA., 1981, 453p.

[Good 84] Good, Michael D. et. al., "Building a User-Derived Interface", Communications of the ACM, Vol. 27, No. 10, pp. 1032- 1043, 1032-1043, 1984.

[Green 76] Green, Thomas R. G. (Editor), The Psychology of Computer Use, Academic Press, New York, NY, 1976, 225p.

[Grimes 75] Grimes, Joe, Network Grammars, pp. 47-83, Summer Institute of Linguistics Publications, Norman, OK, 1975.

[Grishman 84] Grishman, Ralph, "Natural Language Processing", Journal of the American Society for Information Science, Vol. 35 No. 5, pp. 291-296, 1984.

[Hall 85] Hall, Philip P., The Design and Implementation of PC/MISI, a Multiple Information System Interface, Master's Thesis, Computer Science Department, University of Southwestern Louisiana, Lafayette, LA, 1985.

[Hendrix 81] Hendrix, G. and Carbonnel, John G, "A Tutorial On Natural Language Processing", Proceedings of ACM'81, Vol. 1, pp. 4-9, 1981.

[Honeywell_76] (Editor), Multics Relational Data Store (MRDS) Reference Manual, Honeywell Information Systems Publication, Minneappolis, MN, 1976.

[Hunter 81] Hunter, Robin, The Design and Construction of Compilers, pp. 18-56, John Wiley and Sons, Chichester, England, 1981.

[INTELLECT, 85] (Editors), INTELLECT Technical Reference Manual. Artificial Intelligence Corporation, Waltham, MA., 1985.

[Jones 83] Jones, Karen Spark and Yorick Wilks, Automated Natural Language Parsing, John Wiley and Sons, New York, NY, 1983, 210p.

[Katz 72] Katz, Jerrorld J., Semantic Theory, Harper and Row Publishing Company, New York, NY, 465p., 1972.

[Kerningham 76] Kerningham, Brian and Richie, Dennis, The C Programming Language, Technical Report, Bell Laboratories, Murray Hill, NJ, 1976.

[Kerningham 79] Kerningham, Brian, and Pike, J., The UNIX Programming Environment Academic Press, New York, 283p.

[Kidder 82] Kidder, Tracy, Soul of a New Machine, Avon Books, New York, NY, 315p., 1982.

[Lehnert 78] Lehnert, Wendel G., The Process of Question Answering: A Computer Simulation of Cognition, Lawrence Erlbaum Associates, Publishers, New York City, NY, 1978.

[Lesk 76] Lesk, M. E., and E. Schmidt, Lex: A Lexical Analyzer Generator, 12p., Bell Telephone Laboratories, Murray Hill, NJ, 1976.

[Logsdon 80] Logsdon, Thomas S., Computers and Social Controversy, Computer Science Press Inc., Potomac, MD, 1980, 396p.

[Marcus 82] Markus, Michael P., A Theory of Syntactic Recognition for Natural Languages, pp. 221-239, MIT Press, Cambridge, MA, 1982.

[McCalla 83] McCalla, Gordon and Cercone, Nick, "Approaches to Knowledge Representation", IEEE Software Magazine, Vol. 18 No. 10, pp. 12-16, 1983.

[Morrison 84] Morrison, Perry R., "A Survey of Attitudes toward Computers", Communications of the ACM, Vol. 26 No. 12, pp. 1051-1057, 1983.

[Mylopoulos 76] Mylopoulos, John et. al, "TORUS: A Step Toward Bridging the Gap Between Data Bases and the Casual User", Management Systems, Vol. 2, pp. 49-63, Pergamon Press, England, 1976.

[Pratt 83] Pratt, Terence, The Design and Implementation of Programming Languages, Academic Press, New York, 1984, 560p.

[Rich 83] Rich, Elaine, Artificial Intelligence, McGraw-Hill Publications, New York, 1983, 436p.

[Salton 83] Salton, Gerald and McGill, Michael J., Introduction to Modern Information Retrieval, McGraw-Hill Publishing Co., New York City, NY, 1983.

[Sommerville 82] Sommerville, Ian, Software Engineering, Addison Wesley Publishing Co., London, England, 290p, 1980.

[Stonebraker 76] Stonebraker, Michael et. al., The Design and Implementation of INGRES, 73p., Department of Computer Science, UC-Berkeley, Berkeley, CA, 1976.

[Taylor 84] Taylor, Jared, "Putting a Ph.D in your PC", PC Magazine, No 2, February 1984, pp. 167-174, 1984.

[Tennant, 80] Tennant, Harry, Evaluation of Natural Language Processing Systems, Ph. D. Dissertation, University of Illinois at Urbana, Urbana, IL., 1980

[Tennant 81] Tennant, Harry, Natural Language Processing: an Introduction to an Emerging Technology, Petrocelli Books, New York City, NY, 1981.

[Teory 82] Teory, Toby J., and James P. Fry, Design of Database Structures, Prentice-Hall Inc., Englewood Hills, NJ, 490p., 1982.

[Turner 84] Turner, Ray, Software Engineering Methodology, Reston Publishing Co., Reston, VA, 226p., 1984.

[Ullman 82] Ullman, Jefferey D., Principles of Database Systems Computer Science Press Inc., Rockville, ML, 1983.

[Urban 84] Urban, Joseph E., Software Design Methodology Class Notes, University of Southwestern Louisiana, Lafayette, LA, 1984.

[Warnier 79] Warnier, Dominique Jean, Logical Construction of Systems, Van Nostrand Reinhold Co., New York, 1979.

[Wasserman 85] Wasserman, Kenneth, "Physical Object Representation and Generalization: A Survey of Programs for Semantics-Based Natural Language Processing", AI Magazine, Winter 1985.

[Weizenbaum 66] Weizenbaum, J., "Eliza: A Computer Program for the Study of Natural Language Communication between Man and Machine", Communications of the ACM, No. 9, September 1966, pp. 36-45, 1966.

[Wiederhold 84] Wiederhold, Gio, "Knowledge and Database Management", IEEE Software Magazine, Vol. 19 No. 1, pp. 63-73, 1984.

[Wiederhold 77] Wiederhold, Gio, Database Design, Mc-Graw-Hill Publishing Co., New York, NY, 656p, 1977.

[Wilks 82] Wilks, Yorick, Some Thoughts on Procedural Semantics Laurence Erlbaum Associate Publishers, Hillside, NJ, 1982.

[Winograd 83] Winograd, Terry, Language as a Cognitive Process, Addison-Wesley Publishing Co., Reading, MA, 1983.

[Winston 81] Winston, Patric Henry and B. K. P. Horn, LISP, Addison-Wesley Publishing Co., Reading, MA, 1981, 425p.

[Zloof 75] Zloof, M. M. "Query By Example", Proceedings of the National Computer Conference '76, May 1975.

APPENDIX A

SAMPLE DATABASE AND KNOWLEDGE BASE

I. SAMPLE DATABASE

student relation

name	studentid	major	gpa	classi	credit
Collins Philip Y.	225-8770-89	ARCH	2.988	1	30
Diaz Bartholomew	000-8765-22	ELEE	3.874	4	140
Doe Jonathan T.	225-5437-63	ENGL	2.001	9	0
Hellden Mary K.	656-8787-88	HIST	3.586	4	98
Jameson Andrea	999-3431-22	HIST	2.988	3	70
Markowitz Leonid	300-4567-65	CMPS	3.250	1	40
Robinson Smoky	123-5678-90	MUSI	3.780	4	127
Sokky Dianna	021-1872-33	STAT	3.345	1	20
Silver John Long	000-4076-65	CIVE	3.510	4	130
Work Will You	000-0000-01	HIST	1.599	2	35
Ding Ping Sing	255-3565-00	CMPS	3.258	5	3

course relation

dept	number	instructor	descript	credit
CIVE	325	Bauhaus Erich V.	Architectural Design IV.	6
CMPS	150	Jackson Michael	Introduction to CS Majors	3
CMPS	250	Kolf Dieter	Program Design I.	3
CMPS	351	Jackson Michael	Assembly Language	3
ENGL	111	Wallash Tina	English for Others	9
ENGL	699	Cox John A.	English Dissertation	v
FIAR	320	DaVinchi Leonardo	Introduction to the Arts	3
HIST	120	Gentry John A.	History of History	2
HIST	650	Hunn Attilas I.	Invasion and Disaster VI.	9
MATH	111	Wright Wilbur	Introduction to ABC	12
MATH	590	Turing Alan G.	Master's Project	v
MUSI	102	Premoli Flavio A.	Modern Italian Music	2

faculty relation

name	address	ssn	salary
Bauhaus Erich V.	52 Hauss St. Berlin GDR	999-9898-12	36000
Cox John A.	Here Avenue #2 Orange TX	666-9899-89	20000
DaVinchi Leonardo	1200 Plaza Angelo Roma Italy	000-0000-03	42000
Gentry John A.	500 E. 16th St. Opelousas LA.	255-6556-79	18000
Hunn Attilas I.	1 Mongolia Apts Houma LA	000-0000-99	28000
Jackson Michael	114 North St. Lafayette LA	123-4567-89	24000
Premoli Flavio A.	23 Via Rose Milano Italy	232-9998-98	18000
Turing Alan G.	1 Tape Dr. Richmond VA	226-9898-03	40000
Kolf Dieter	Box 14622 Broussard LA	544-5689-00	22500
Wallash Tina	45 Oak Bvd. Hammond LA	559-9999-99	24000
Wright Wilbur	62 Main St. Baton Rouge LA	222-9986-66	32000

records relation

dept	number	name	date	grade
CIVE	325	Collins Philip Y.	1982	D
CIVE	325	Silver John Long	1981	WB
CIVE	325	Silver John Long	1980	B
CMPS	250	Work Will You	1982	NR
CMPS	351	Diaz Bartholomew	1982	A
CMPS	351	Ding Ping Sing	1982	A
CMPS	405	Diaz Bartholomew	1983	A
CMPS	405	Ding Ping Sing	1983	B
ENGL	111	Doe Jonathan T.	1982	F
ENGL	111	Markowitz Leonid	1981	D
ENGL	111	Sokky Diana	1980	C
ENGL	111	Silver John Long	1982	A
ENGL	111	Work Will You	1981	F
ENGL	111	Ding Ping Sing	1984	WF
ENGL	699	Work Will You	1984	A
FIAR	320	Collins Philip Y.	1981	D
FIAR	320	Robinson Smoky	1981	B
FIAR	320	Silver John Long	1982	C
FIAR	320	Work Will You	1985	F
FIAR	320	Ding Ping Sing	1984	B
HIST	120	Jameson Andrea	1982	A
HIST	650	Hellden Mary K.	1982	A
HIST	650	Hellden Mary K.	1980	C
HIST	650	Jameson Andrea	1985	A
MATH	111	Diaz Bartholomew	1980	B
MATH	111	Markowitz Leonid	1981	F
MATH	111	Ding Ping Sing	1982	D
MUSI	102	Robinson Smoky	1980	A
MUSI	102	Robinson Smoky	1980	WA
STAT	454	Markowitz Leonid	1984	A
STAT	454	Sokky Diana	1981	A
STAT	454	Ding Ping Sing	1982	C
STAT	521	Markowitz Leonid	1985	W
STAT	521	Sokky Diana	1982	A
STAT	521	Ding Ping Sing	1983	A

II. SAMPLE KNOWLEDGE BASE

dictionary

entry	type
address	n
course	n
gpa	n
grade	n
instructor	n
major	n
name	n
number	n
salary	n
ssn	n
student	n
earn	v
retrieve	v
live	v
work	v
study	v
that	s
this	s
out	s
in	s
and	b
greater	r
: : :	:
: : :	:

sequence relation

word	pno	rank
social	1	1
security	1	2
number	1	3
student	2	1
id	2	2
number	2	3
student	3	1
id	3	2
: : : :	: :	: :
: : : :	: :	: :

synonym relation

```
|sterm     |sreplace                |
|-----------------------------------|
|I         |.                       |
|ME        |.                       |
|YOU       |.                       |
|PLEASE    |.                       |
|THE       |.                       |
|take      |study                   |
|&         |and                     |
|||        |or                      |
|>=        |greatereq               |
| : : :    | : : : : :              |
```

adject relation

```
|aterm     |anoun     |aproperty                         |
|-------------------------------------------------------|
|good      |student   |gpa > 3.000                       |
|bad       |student   |gpa < 2.000                       |
|rich      |faculty   |salary > 40000                    |
|poor      |faculty   |salary < 20000                    |
|: : : :   | : : : :  | : : : : : : :                    |
|          |          |                                  |
```

noun relation

```
|nterm     |ndbpro|
|-----------------|
|address   |a     |
|classif   |a     |
|course    |r     |
|credit    |a     |
|major     |a     |
|name      |a     |
|records   |r     |
|salary    |a     |
|ssn       |a     |
|student   |r     |
|: : : : : |:     |
|: : : : : |:     |
```

verb relation

vterm	vobject	vsubject
earn	faculty	salary
live	faculty	address
work	faculty	dept
take	student	dept
take	student	number
make	student	grade
teach	faculty	number

frame relation

relname	attrname	pattern	rangelow	rangehi	op	type
faculty	name	*	-	-	t	char
faculty	address	*	-	-	t	char
faculty	ssn	[0-9]*-[0-9]*-[0-9]*	000-0000-00	999-9999-99	t	char
faculty	salary	[0-9]*.[0-9]*	0.0	99.999	a	num
course	dept	[A-Z][A-Z][A-Z][A-Z]	-	-	t	char
course	number	[1-8][1-9][1-9]	100	699	n	num
course	instructor	*	-	-	t	char
course	descript	*	-	-	t	char
course	credit	[0-6]	0	6	n	num
student	name	*	-	-	t	char
student	studentid	[0-9]*-[0-9]*-[0-9]*	000-0000-00	999-9999-99	t	char
student	major	[A-Z][A-Z][A-Z][A-Z]	-	-	t	char
student	gpa	[0-9].[0-9]*	0.000	4.000	a	num
student	classif	[0-6]	0	6	n	num
student	major	[0-9]*	0	200	n	num
records	dept	[A-Z][A-Z][A-Z][A-Z]	-	-	t	char
records	number	[1-8][1-9][1-9]	100	699	n	num
records	name	*	-	-	t	char
records	date	[0-9]*	1975	1985	n	num
records	grade	[A-Z]	A	Z	t	char

APPENDIX B

SAMPLE SESSIONS OF KARL USAGE

```
% karl

     The Knowledge Assisted Retrieval Language

                   Version 1.02

> give me the names and student numbers for students

  studying "STAT"

|name                |studentid   |
|---------------------------------|
|Sokky Dianna        |021-1872-33|
|---------------------------------|
(1 tuple)

> who is "000-4076-65" ?

*** Ambiguity: attribute 'name' belongs to relationships:

  1  student
  2  faculty

*** Please select value from  1 to  2: 1

|name                |studentid   |major |gpa   |classi|credit|
|--------------------------------------------------------------|
|Silver John Long    |000-4076-65|CIVE  |3.510|4     |   130|
|--------------------------------------------------------------|
(1 tuple)

> retrieve the names of the rich faculty

|name                |
|--------------------|
|DaVinchi Leonardo   |
|--------------------|
> give me all the courses in department "CMPS" or "FIAR"

|dept |number|instructor          |description                 |credit|
|--------------------------------------------------------------------|
|CMPS |150   |Jackson Michael     |Introduction to CS Majors   |3     |
|CMPS |250   |Kolf Dieter         |Program Design I.           |3     |
|CMPS |351   |Jackson Michael     |Assembly Language           |3     |
|FIAR |320   |Davinchi Leonardo   |Introduction to the Arts    |3     |
|--------------------------------------------------------------------|
(4 tuples)

> give me the rich students

*** Error: Attribute 'salary' not associated with

           relation 'student'

    Query aborted

> from the students in "CMPS251", who has a gpa of

  more than "2.000"?

*** Error: Could not parse input sentence

           Syntax  error. No such sentence type supported.

    Query aborted

> show me the records of "1982"

|dept  |number|name                |date |grade |
|-----------------------------------------------|
|CIVE  |325   |Collins Philip Y.   |1982 |D     |
|CMPS  |250   |Work Will You       |1982 |NR    |
|CMPS  |351   |Diaz Bartholomew    |1982 |A     |
|CMPS  |351   |Ding Ping Sing      |1982 |A     |
|ENGL  |111   |Doe Jonathan T.     |1982 |F     |
|ENGL  |111   |Silver John Long    |1982 |A     |
|FIAR  |320   |Silver John Long    |1982 |C     |
|HIST  |120   |Jameson Andrea      |1982 |A     |
|HIST  |650   |Hellden Mary K.     |1982 |A     |
|MATH  |111   |Ding Ping Sing      |1982 |D     |
|STAT  |454   |Ding Ping Sing      |1982 |C     |
|STAT  |521   |Sokky Diana         |1982 |A     |
|-----------------------------------------------|
(12 tuples)
```

```
> give the_transcript for "Ding Ping Sing"

|dept  |number|name                |date  |grade |
|------------------------------------------------|
|CMPS  |351   |Ding Ping Sing      |1982  |A     |
|CMPS  |405   |Ding Ping Sing      |1983  |B     |
|ENGL  |111   |Ding Ping Sing      |1984  |WF    |
|FIAR  |320   |Ding Ping Sing      |1984  |B     |
|MATH  |111   |Ding Ping Sing      |1982  |D     |
|STAT  |454   |Ding Ping Sing      |1982  |C     |
|STAT  |521   |Ding Ping Sing      |1983  |A     |
|------------------------------------------------|
(7 tuples)

> who is living in "52 Hauss St. Berlin GDR"

|name                |
|--------------------|
|Bauhaus Erich V.    |
|--------------------|
(1 tuple)

> give me the names and student id. numbers of the

  good students

|name                |studentid  |
|--------------------------------|
|Diaz Bartholomew    |000-8765-22|
|Hellden Mary K.     |656-8787-88|
|Robinson Smoky      |123-5678-90|
|Silver John Long    |000-4076-65|
|--------------------------------|
(4 tuples)

> retrieve students "Hellden Mary K." or "Silver John Long"

|name                |studentid  |major |gpa           |classi|credit|
|--------------------------------------------------------------------|
|Hellden Mary K.     |656-8787-88|HIST  |         3.586|4     |    98|
|Silver John Long    |000-4076-65|CIVE  |         3.510|4     |   130|
|--------------------------------------------------------------------|
(2 tuples)
```

```
> give me all the grades during "1982" or "1983"

|grade |
|------|
|D     |
|NR    |
|A     |
|A     |
|A     |
|B     |
|F     |
|A     |
|C     |
|A     |
|A     |
|D     |
|C     |
|A     |
|A     |
|------|
(15 tuples)

> exit

Karl 1.02: Good Bye!
%
```

Triantafyllopoulos, Spiros, B.S., University of Southwestern Louisiana, Fall 1983
Master of Science, Summer 1985
Major: Computer Science
Title of Thesis: KARL: A Knowledge Assisted Retrieval Language
Thesis Directed by Professor Wayne D. Dominick
Pages in Thesis, 146; Words in Abstract, 214

ABSTRACT

Data classification and storage are tasks typically performed by application specialists. In contrast, information users are primarily non-computer specialists who use information in their decision-making and other activities. Interaction efficiency between such users and the computer is often reduced by machine requirements and resulting user reluctance to use the system.

This thesis examines the problems associated with information retrieval for non-computer specialist users, and proposes a method for communicating in restricted English that uses knowledge of the entities involved, relationships between entities, and basic English language syntax and semantics to translate the user requests into formal queries. The proposed method includes an intelligent dictionary, syntax and semantic verifiers, and a formal query generator. In addition, the proposed system has a learning capability that can improve portability and performance.

With the increasing demand for efficient human-machine communication, the significance of this thesis becomes apparent. As human resources become more valuable, software systems that will assist in improving the human-machine interface will be needed and research addressing new solutiaddressings will be of upmost importance. This thesis presents an initial design and implementation as a foundation for further research and development into the emerging field of natural language database query systems.

BIOGRAPHICAL SKETCH

Spiros Triantafyllopoulos was born in [redacted], in [redacted] [redacted] [redacted]. He studied Civil Engineering in the Center for Higher Technical Education in Pireaus, Greece, and received his B.Sc. in Computer Science in December 1983 from the University of Southwestern Louisiana. He attended the University as a graduate student from January 1984 to August 1985, receiving his Master of Science in Computer Science. Mr. Triantafyllopoulos has joined the staff of the Computer Science Department, General Motors Research Laboratories in Warren, Michigan, as a Technology Transfer Scientist, working in R&D on integrated software environments. Previous publications include:

"Monitor Design" and "Monitor Implementation", Chapters 6 and 7 in Information System Monitoring, Analysis, and Evaluation, W. D. Dominick and W. D. Penniman, book to be published by John Wiley and Sons Inc., 1985.

"Knowledge-Based Information Retrieval: Techniques and Applications", Proceedings of the 1985 ACM Thirteenth Annual Computer Science Conference, March 12-14, 1985.

"PC-Based Research and Development for Information Storage and Retrieval Systems Support," with Frank Y. Chum, Dennis R. Moreau and Philip P. Hall, Proceedings of the Eighteenth Annual Hawaii Conference on Systems Sciences, Honolulu, Hawaii, January 2-4, 1985, Vol. II, pp. 789-797.

"General Specifications for the Development of a PC-based Simulator of the NASA RECON System," USL/DBMS NASA/PC R&D Working Paper Series Report Number DBMS.NASA/PC R&D-4, August 2, 1984, 21p.

"General Specifications for the Development of a USL NASA PC R&D Statistical Analysis Support Package," with Jinous Bassari, USL/DBMS NASA/PC R&D Working Paper Series Report Number DBMS.NASA/PC R&D-5, August 2, 1984, 14p.

"The USL NASA PC R&D Project: Detailed Specifications of Objectives," with Frank Y. Chum, Philip P. Hall, and Dennis R. Moreau, USL/DBMS NASA/PC R&D Working Paper Series Report Number DBMS.NASA/PC R&D-8, August 15, 1984, 21p.

"A Performance Evaluation of the IBM 370/XT Personal Computer," USL/DBMS NASA/PC R&D Working Paper Series Report Number DBMS.NASA/PC R&D-10, October 5, 1984, 47p.

"IBM PC/IX Operating System Evaluation Plan," with Martin Granier and Philip P. Hall, USL/DBMS NASA/PC R&D Working Paper Series Report Number DBMS.NASA/PC R&D-14, November 28, 1984, 9p.

4.22

1. Report No. IN-82	2. Government Accession No. 183567 ~~147399~~	3. Recipient's Catalog No.
4. Title and Subtitle 158P USL/NGT-19-010-900: KARL: A KNOWLEDGE-ASSISTED RETRIEVAL LANGUAGE		5. Report Date DATE October 31, 1985 OVERRIDE
		6. Performing Organization Code
7. Author(s) SPIROS TRIANTAFYLLOPOULOS		8. Performing Organization Report No.
		10. Work Unit No.
9. Performing Organization Name and Address University of Southwestern Louisiana The Center for Advanced Computer Studies P.O. Box 44330 Lafayette, LA 70504-4330		11. Contract or Grant No. NGT-19-010-900
		13. Type of Report and Period Covered FINAL; 07/01/85 - 12/31/87
12. Sponsoring Agency Name and Address		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

Data classification and storage are tasks typically performed by application specialists. In contrast, information users are primarily non-computer specialists who use information in their decision-making and other activities. Interaction efficiency between such users and the computer is often reduced by machine requirements and resulting user reluctance to use the system. This thesis examines the problems associated with information retrieval for non-computer specialist users, and proposes a method for communicating in restricted English that uses knowledge of the entities involved, relationships between entities, and basic English language syntax and semantics to translate the user requests into formal queries. The proposed method includes an intelligent dictionary, syntax and semantic verifiers, and a formal query generator. In addition, the proposed system has a learning capability that can improve portability and performance. With the increasing demand for efficient human-machine communication, the significance of this thesis becomes apparent. As human resources become more valuable, software systems that will assist in improving the human-machine interface will be needed and research addressing new solutions will be of utmost importance. This thesis presents an initial design and implementation as a foundation for further research and development into the emerging field of natural language database query systems.

This report represents one of the 72 attachment reports to the University of Southwestern Louisiana's Final Report on NASA Grant NGT-19-010-900. Accordingly, appropriate care should be taken in using this report out of the context of the full Final Report.

17. Key Words (Suggested by Author(s)) KARL, Knowledge-Assisted Retrieval Language, Information Storage and Retrieval Systems	18. Distribution Statement		
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 146	22. Price*

*For sale by the National Technical Information Service, Springfield, Virginia 22161